F.Y.B.Sc.IT –SEM II

OBJECT ORIENTED PROGRAMMING WITH C++ (PUSIT206T)

By,
Prof. Nikita Madwal

UNIT 2

5. FUNCTION IN C++
6. CLASS AND OBJECTS IN C++
7. WORKING WITH CONSTRUCTOR AND DESTRUCTOR
8. WORKING WITH OPERATOR OVERLOADING

5. FUNCTION IN C++

Introduction

- Functions can be viewed as basic building blocks of the program.
- A function is a **self-contained program** segment that carries out **some specific, well-defined task**.
- **A function is a group of statements that together perform a task.**
- Every program consists of one or more functions
- Every program has **at least one function**, which is **main()**
- Additional functions will be subordinate to main

- Functions comprises of set of instructions delimited inside by **{ }** braces
- **Types of function**

1. Library function

- Library functions are ready made or they are **built-in functions** made available through C++ library
- These are the **functions** which are **declared in the C++ header files**
- For e.g. strcpy(), strlen()

2. User defined function

- These functions are those **functions** which are **defined** by the **time of writing of program.**
- The functions which are **created by the programmer** is known as **User-defined function**

- **Advantage of functions**
- By using functions, we can **avoid rewriting same logic/code again and again** in a program.
- These functions can be **called multiple times** as and when required, there is no limit in calling functions.
- These functions can be **called from any part of the program**

Function Prototypes (Function Declaration)

- A function prototype is **simply the declaration of a function** that specifies ***function's name, parameters and return type***. It doesn't contain function body.
- This is also called as **function declaration**
- A function prototype gives information to the compiler that the function may later be used in the program
- Function prototypes are usually written at the ***beginning of a program***.
- The ***argument*** are also called as ***parameters***
- Syntax

return_type function_name(data_type argument list,.....);

Accessing A Function (Function Call)

- A **function can be accessed** (i.e., called) by **specifying its name**, followed by a **list of arguments** enclosed in parentheses and separated by commas.
- This is also called **function call**
- It is **called inside a program whenever it is required to call a function**. It is **only called by its name** in the **main() function** of a program
- Syntax

  function_name(argument1, argument2, ...);

Defining Function (Function Definition)

- **Function definition** contains the **block of code** to perform **a specific task.**
- The **actual task** of the function is **implemented in the function definition**
- The function definition is also known as the **body of the function**
- Syntax

```
return_type   function_name(data_type parameter,…..)
 {
      function body (i.e. code to be executed);
 }
```

- A **function definition** contains the parts as follow:

❖**Return Data_Type:**

- It **defines the return data type of a value in the function**.
- The return data type can be integer, float, character, etc.
- Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**

❖**Function Name**

- It defines the **actual name of a function** that contains some parameters.

❖**Parameters/ Arguments**

- It is a parameter that **passed inside the function name** of a program.
- Parameters can be any type, order, and the number of parameters.

❖**Function Body**

- It is the **collection of the statements to be executed** for performing the **specific tasks in a function**.

- Depending upon parameters and return type **functions are classified into four categories** :

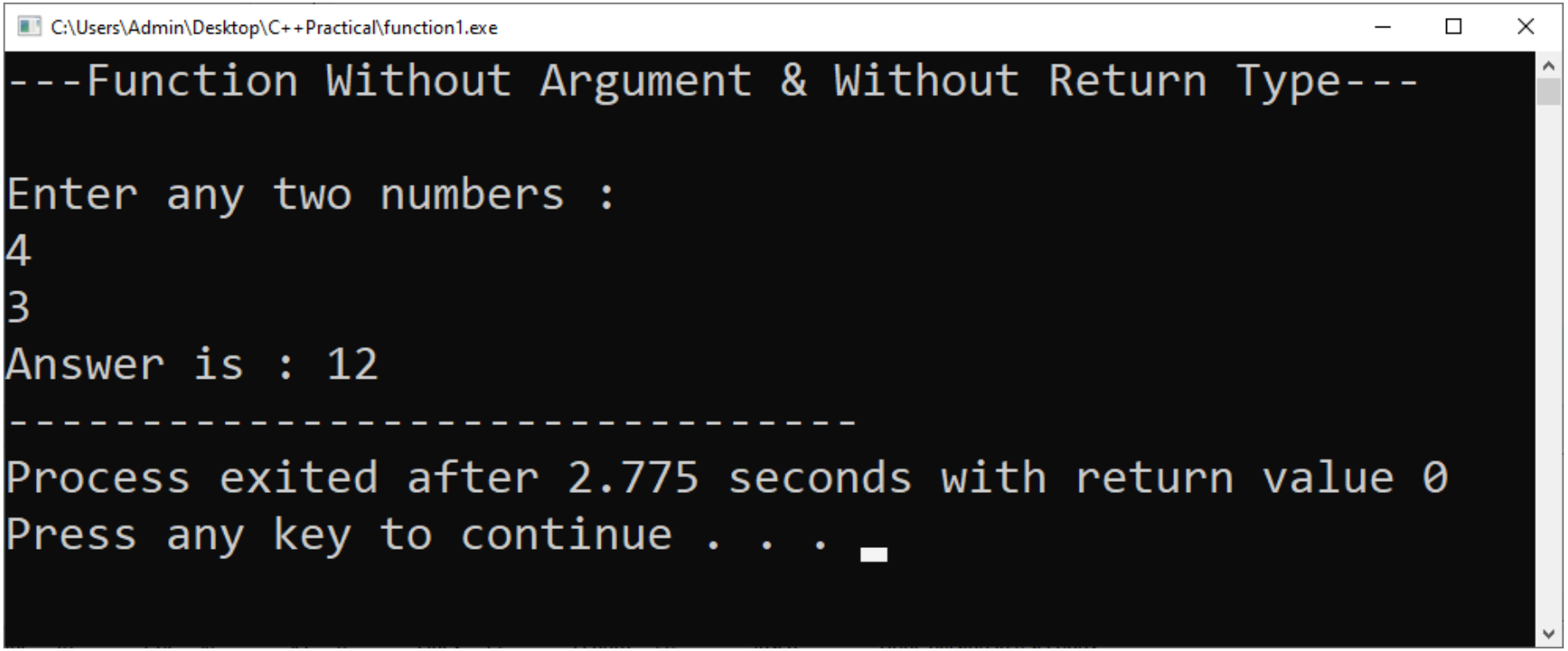
1. Function without argument and without return value
2. Function without argument and with return value
3. Function with argument and without return value
4. Function with argument and with return value

Function without argument and without return value

function1.cpp

```cpp
#include<iostream>
using namespace std;
void multi(); //function declaration

int main()
{
    cout<<"---Function Without Argument & Without Return Type---";
    multi(); //function call
    return 0;
}

void multi() //function definition
{
    int p,q,ans;
    cout<<"\n\nEnter any two numbers :\n";
    cin>>p>>q;
    ans=p*q;
    cout<<"Answer is : "<<ans;
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\function1.exe
---Function Without Argument & Without Return Type---
Enter any two numbers :
4
3
Answer is : 12

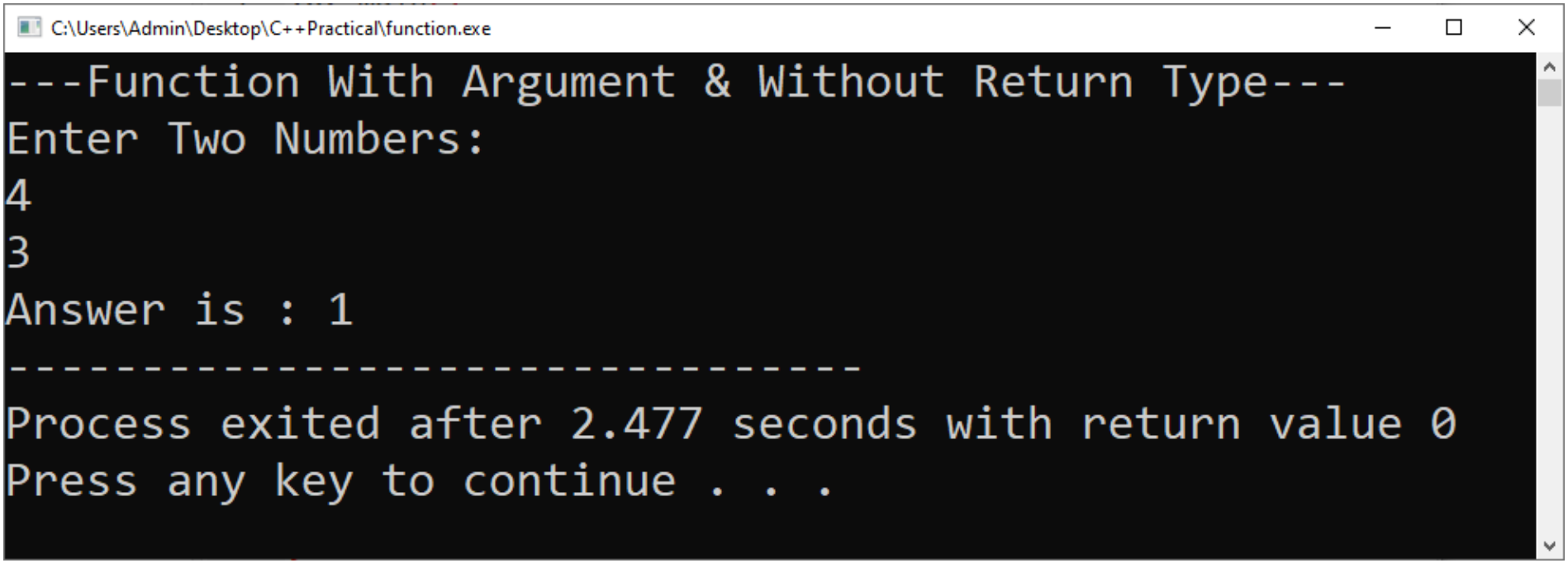
Process exited after 2.775 seconds with return value 0
Press any key to continue . . .

Function with argument and without return value

function.cpp

```cpp
#include<iostream>
using namespace std;
void sub(int x,int y);
int main()
{
    int a,b;
    cout<<"---Function With Argument & Without Return Type---";
    cout<<"\nEnter Two Numbers: \n";
    cin>>a>>b;
    sub(a,b);
    return 0;
}

void sub(int a,int b)
{
    int result;
    result=a-b;
    cout<<"Answer is : "<<result;
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\function.exe
---Function With Argument & Without Return Type---
Enter Two Numbers:
4
3
Answer is : 1

Process exited after 2.477 seconds with return value 0
Press any key to continue . . .

Recursion

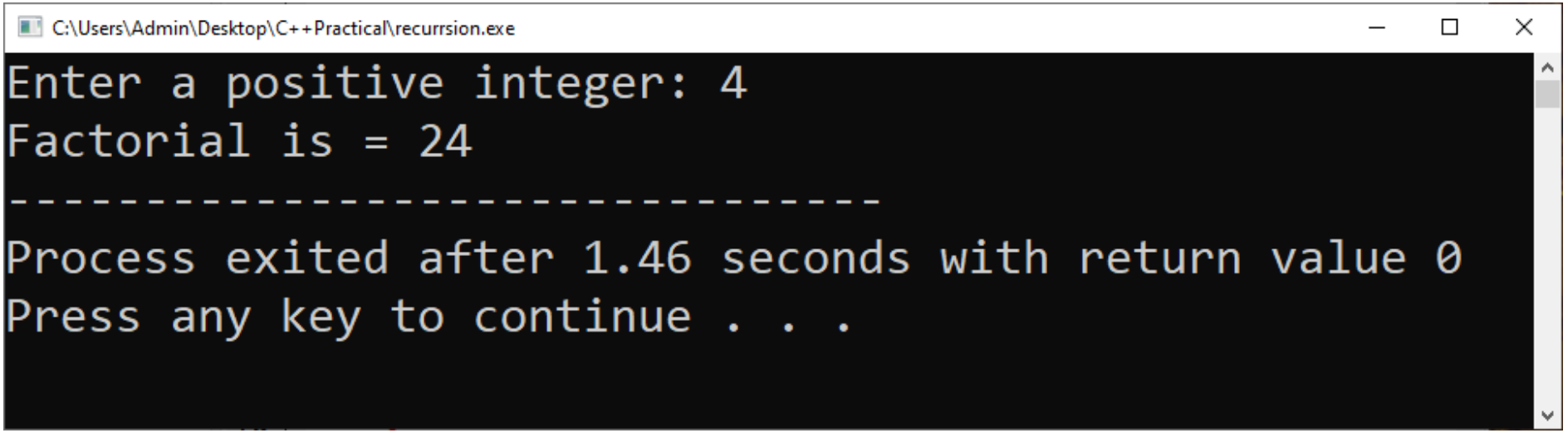
- Recursion is a **programming technique** in which a **function calls itself for a number of times until a particular condition is satisfied**.
- Recursion is a process by which a **function calls itself repeatedly**
- The **function** which **calls itself** is called a **recursive function**, and the **function call** is **termed** a **recursive call**

- E.g.

recurrsion.cpp

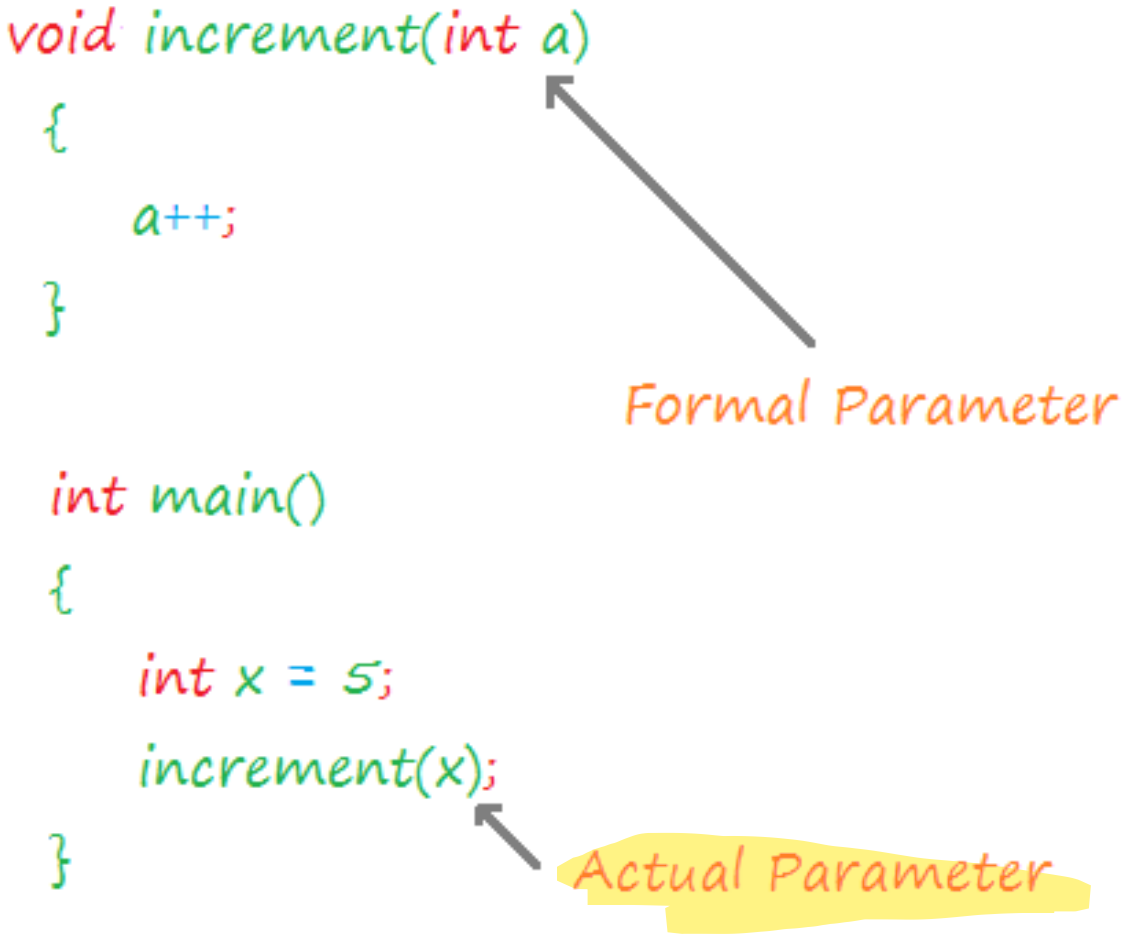
```cpp
#include<iostream>
using namespace std;
int fact(int);
int main()
{
    int n,f;
    cout<<"Enter a positive integer: ";
    cin>>n;
    f=fact(n);
    cout<<"Factorial is = "<<f;
}
int fact(int n)
{
    if (n==0)
    {
        return 0;
    }
    else if(n==1)
    {
        return 1;
    }
    else
        return n*fact(n-1);
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\recurrsion.exe
Enter a positive integer: 4
Factorial is = 24

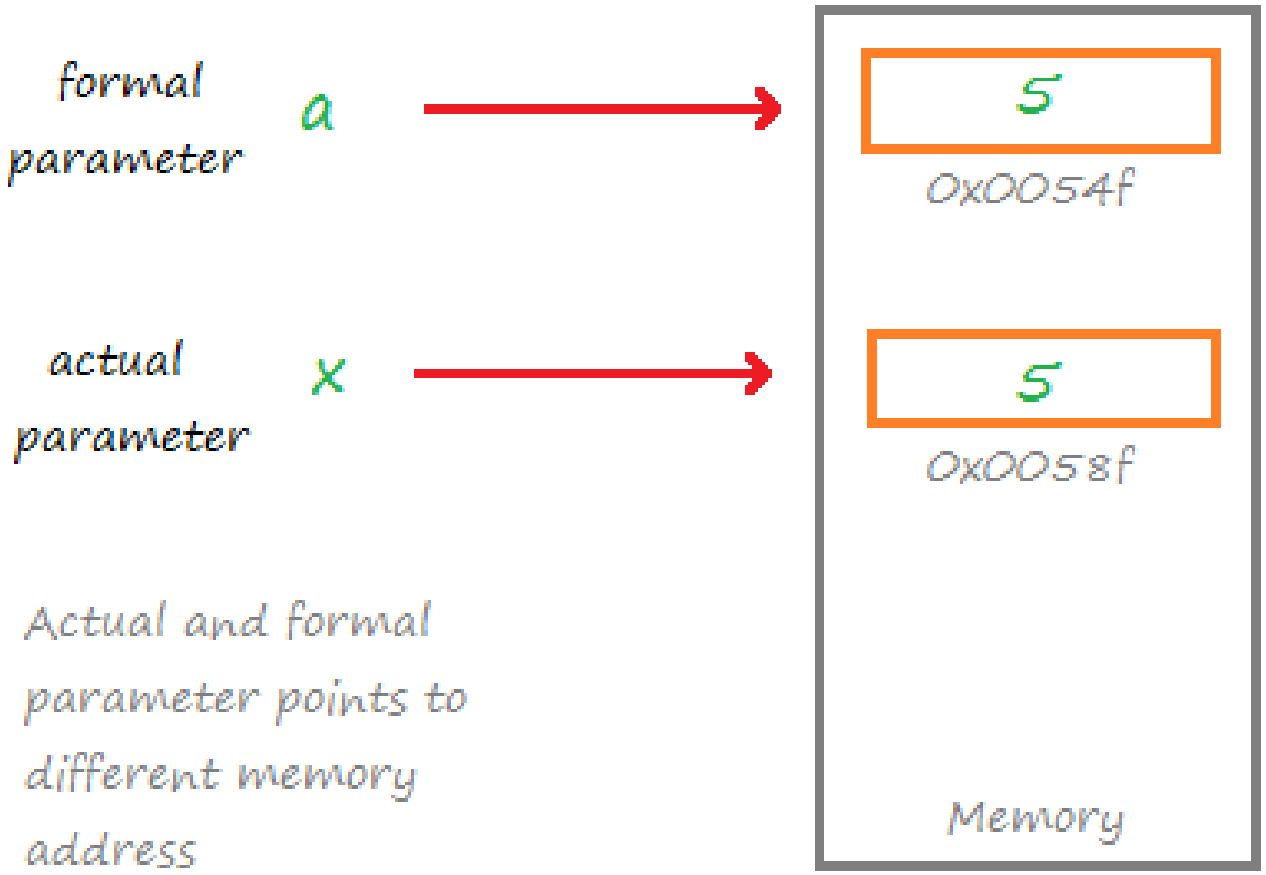
Process exited after 1.46 seconds with return value 0
Press any key to continue . . .

- **Actual Parameters** are the parameters that appear in the function call statement.
- **Formal Parameter**s are the parameters that appear in the declaration of the function which has been called.



Call by value

- The call by value method of passing arguments to a function **copies the actual value of an argument into the formal parameter of the function**
- In this case, **changes made** to the parameter inside the function **have no effect on the argument**. This is **because Both the actual and formal parameters point to different locations in memory.**(they both have different memory addresses)
- Call by value method is useful when we do not want the values of the actual parameters to be changed by the function that has been invoked.



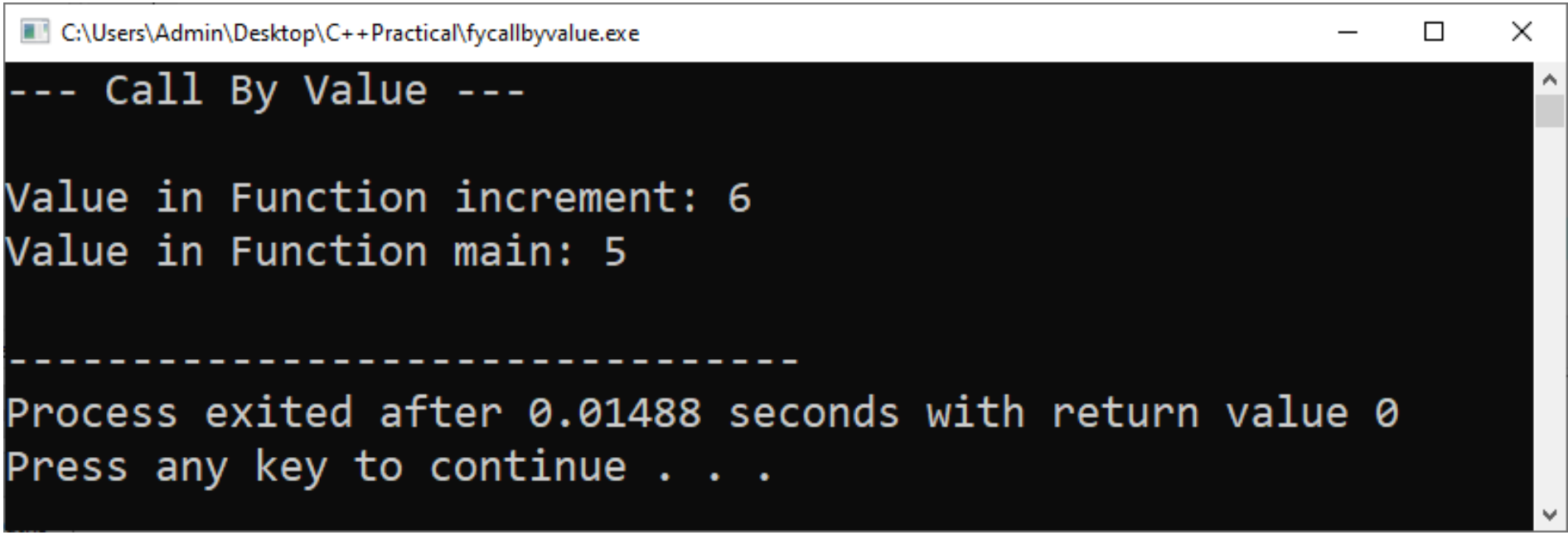
Call by Value in C++

- E.g.

fycallbyvalue.cpp

```cpp
#include <iostream>
using namespace std;
void increment(int a);
int main()
{
    cout<<"--- Call By Value ---"<<endl<<endl;
    int x = 5;
    increment(x);
    cout << "Value in Function main: "<< x <<endl;
    return 0;
}
void increment(int a)
{
    a++;
    cout << "Value in Function increment: "<< a <<endl;
}
```

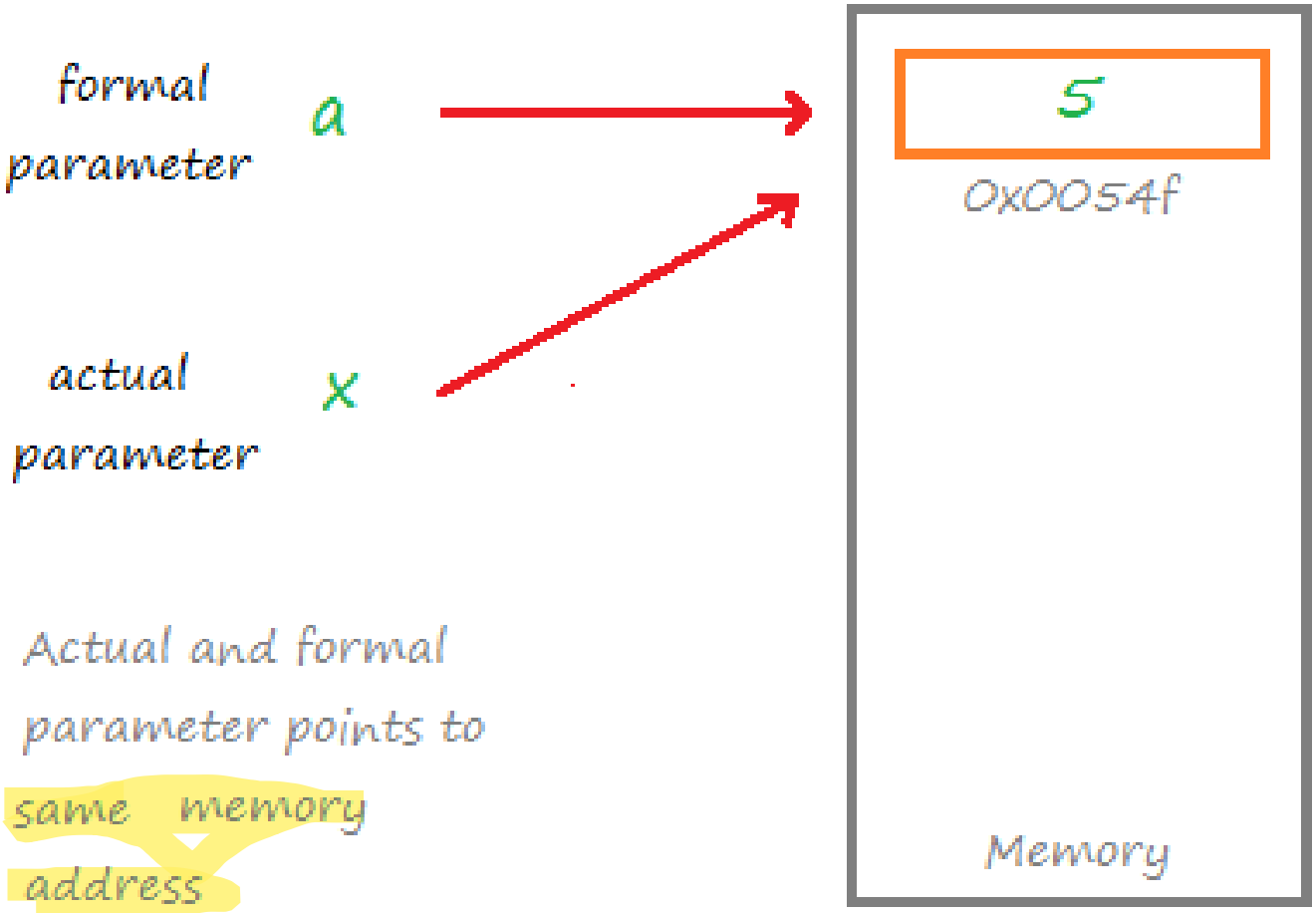
- Output:


C:\Users\Admin\Desktop\C++Practical\fycallbyvalue.exe
--- Call By Value ---
Value in Function increment: 6
Value in Function main: 5

Process exited after 0.01488 seconds with return value 0
Press any key to continue . . .

Call by Reference

- The call by reference method of **passing arguments to a function copies the reference of an argument into the formal parameter.**
- In the call by reference, **both formal and actual parameters share the same value**.
- Both the **actual and formal parameter points** to the **same address in the memory**.
- That means any **change** on one type of parameter **will also be reflected** by other.
- ***Note:*** *For creating reference, the '**&**' operator is used in preceding of variable name.*



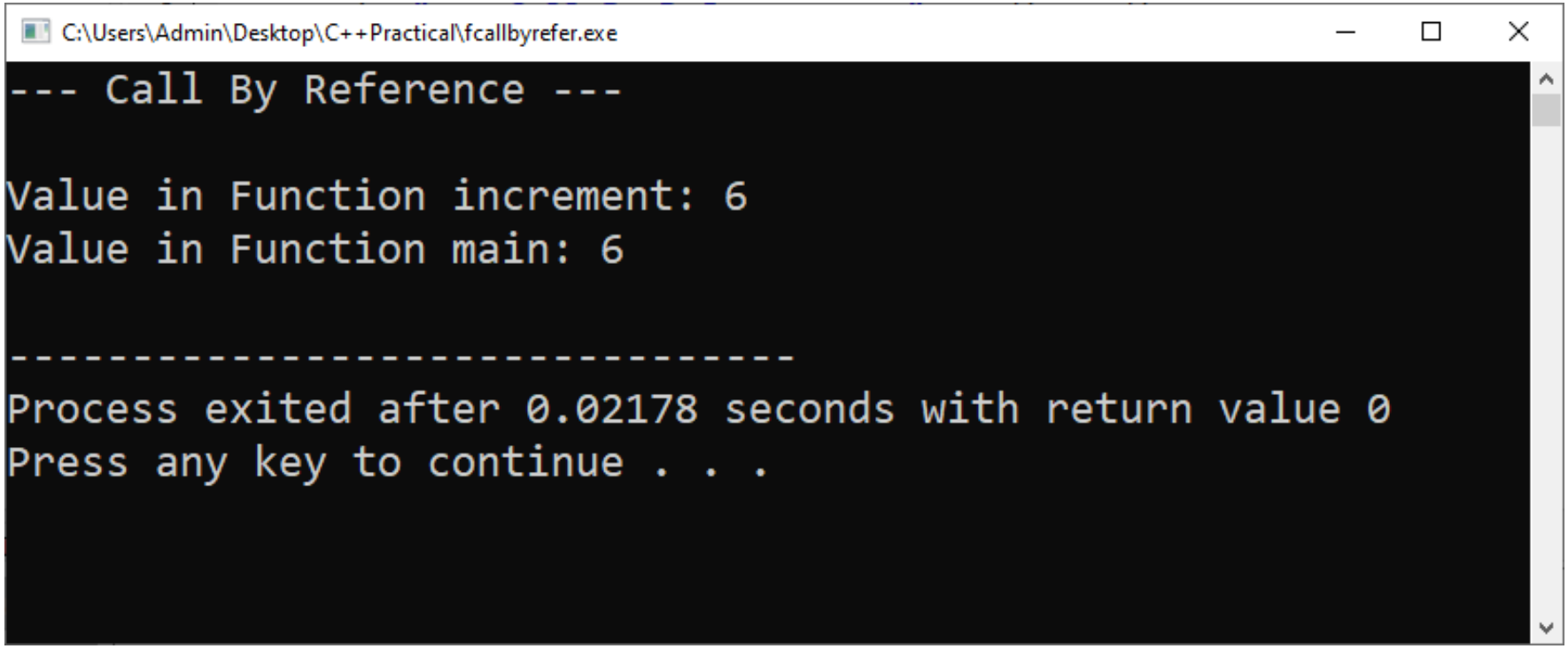
Call by Reference in C++

- E.g.

fcallbyrefer.cpp

```
1   #include <iostream>
2   using namespace std;
3   void increment(int &a);
4   int main()
5   {
6       cout<<"--- Call By Reference ---"<<endl<<endl;
7       int x = 5;
8       increment(x);
9       cout << "Value in Function main: "<< x <<endl;
10      return 0;
11  }
12  void increment(int &a)
13  {
14      a++;
15      cout << "Value in Function increment: "<< a <<endl;
16  }
```

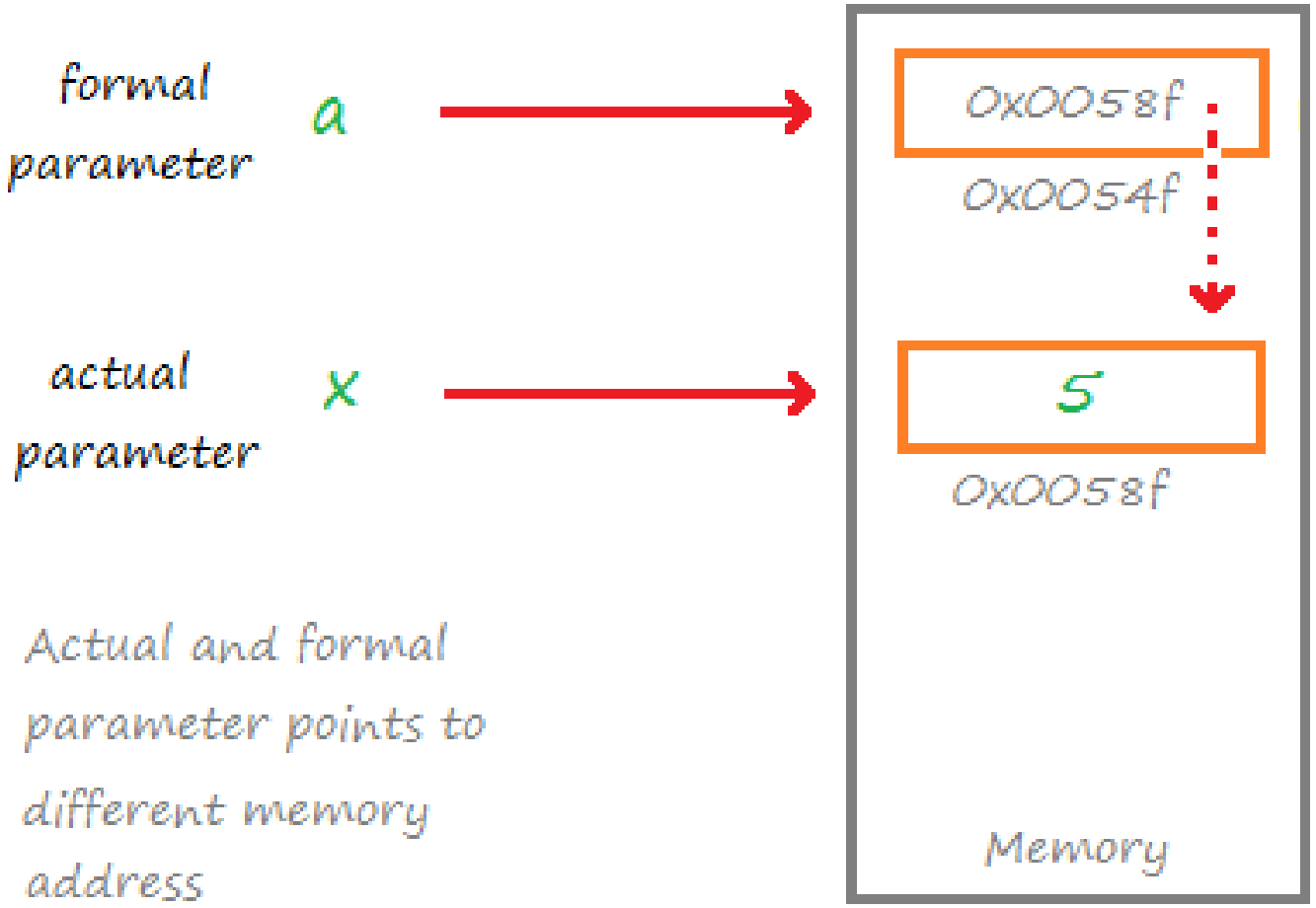
- Output:


C:\Users\Admin\Desktop\C++Practical\fcallbyrefer.exe
--- Call By Reference ---
Value in Function increment: 6
Value in Function main: 6

Process exited after 0.02178 seconds with return value 0
Press any key to continue . . .

Call by Address

- In the call by address method, **both actual and formal parameters indirectly share the same variable**.
- In this type of call mechanism, pointer variables are used as formal parameters.
- The **formal pointer variable holds the address of the actual parameter**, hence the **changes** done by the formal parameter is **also reflected** in the actual parameter.
- In this type , both parameters **point to different locations in memory**, **but** since the **formal parameter stores the address of the actual parameter**, they **share the same value.**



Actual and formal parameter points to different memory address

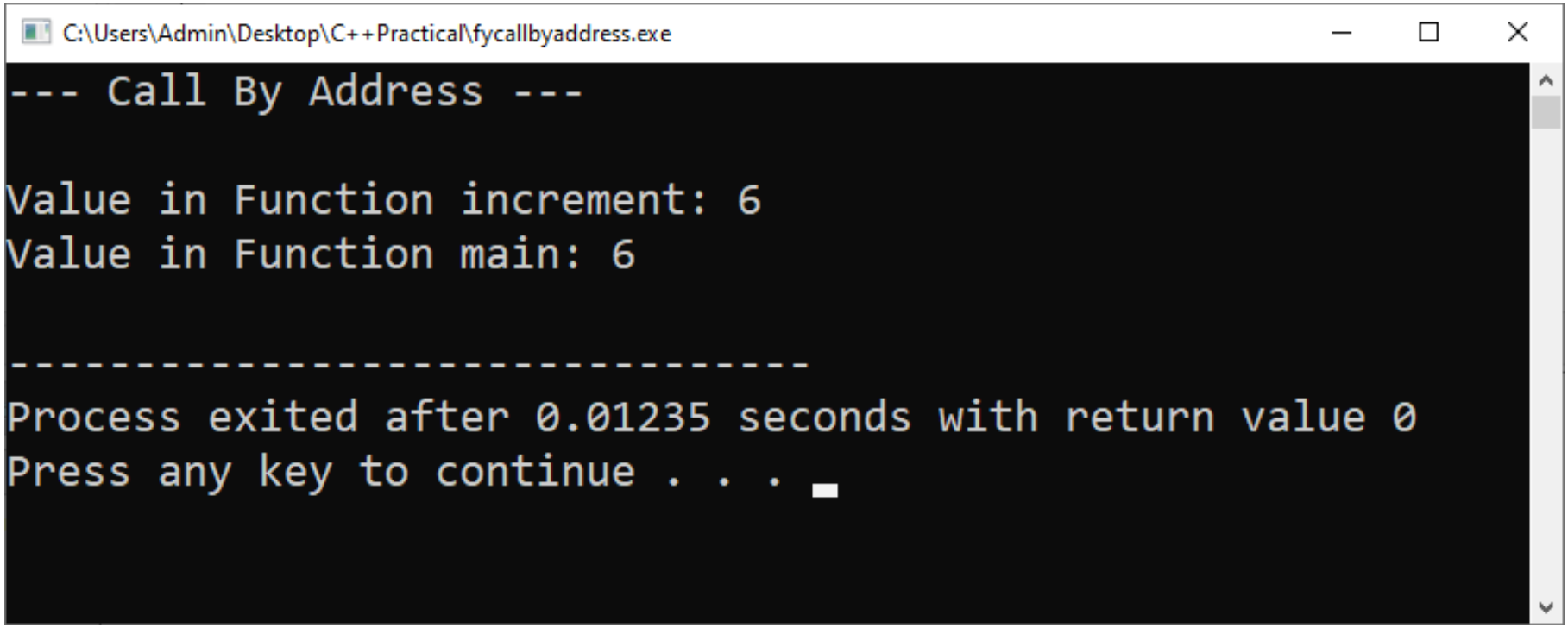
Call by Address in C++

- E.g.

fycallbyaddress.cpp

```
1   #include <iostream>
2   using namespace std;
3    void increment(int *a);
4   int main()
5   {
6       cout<<"--- Call By Address ---"<<endl<<endl;
7       int x = 5;
8       increment(&x);
9       cout << "Value in Function main: "<< x <<endl;
10      return 0;
11  }
12  void increment(int *a)
13  {
14      (*a)++;
15      cout << "Value in Function increment: "<< *a <<endl;
16  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\fycallbyaddress.exe
--- Call By Address ---
Value in Function increment: 6
Value in Function main: 6

Process exited after 0.01235 seconds with return value 0
Press any key to continue . . .

Call by Reference vs Call by Address

Call By Reference

1. The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter
2. Memory is allocated to actual parameter only and formal parameters share that memory

Call By Address

1. In this mechanism, address of the actual arguments are copied to the formal parameters
2. Memory is allocated to both actual and formal parameter

Return by Reference

- In C++ Programming, not only can you pass values by reference to a function but you can also return a value by reference.
- In this mechanism, global variable must be used.
- Syntax:

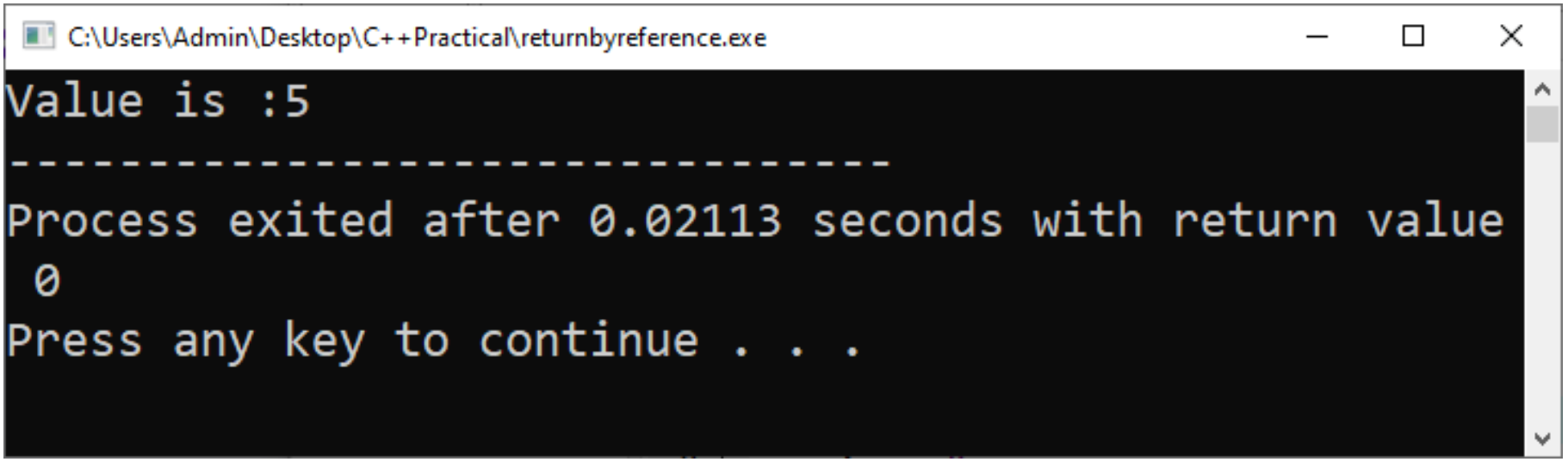
```
return_type & function_name (arguments)
{
    function body;
}
```

- E.g.

returnbyreference.cpp

```
1   #include <iostream>
2   using namespace std;
3   int num;  // global variable
4   int& test(); // function declaration
5   int main()
6   {
7     test() = 5;
8     cout<<"Value is :"<<num;
9     return 0;
10  }
11  int& test()
12  {
13    return num;
14  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\returnbyreference.exe
Value is :5

Process exited after 0.02113 seconds with return value
0
Press any key to continue . . .

- **Note:**
- Ordinary function returns value but this function doesn't. Hence, you cannot return a constant from the function.

```
int& test()
{
    return 2;
}
```

- You cannot return a local variable from this function.

```
int& test()
{
    int n=2;
    return n;
}
```

Inline Function

- **To eliminate the cost of calls and to save execution time in short functions C++ proposes a new feature called inline function.**
- When an instruction of a function call is encountered during the compilation of a program, its memory address is stored by the compiler. The function arguments are copied on the stack and after the execution of the code, the control is transferred to the calling instruction. This process can sometimes cause overhead in function calls. This issue is resolved by using the inline functions.

- The inline functions **overcome** the **overhead and also make the program faster by reducing the execution time of the program.**
- In case of inline functions, the compiler does not go through the above process of switching between the stack and calling function**. Instead, it copies the definition of the inline function and the calling instruction with that definition.**
- Syntax:

```
inline return type function_name (parameters)
{
    function definition;
}
```

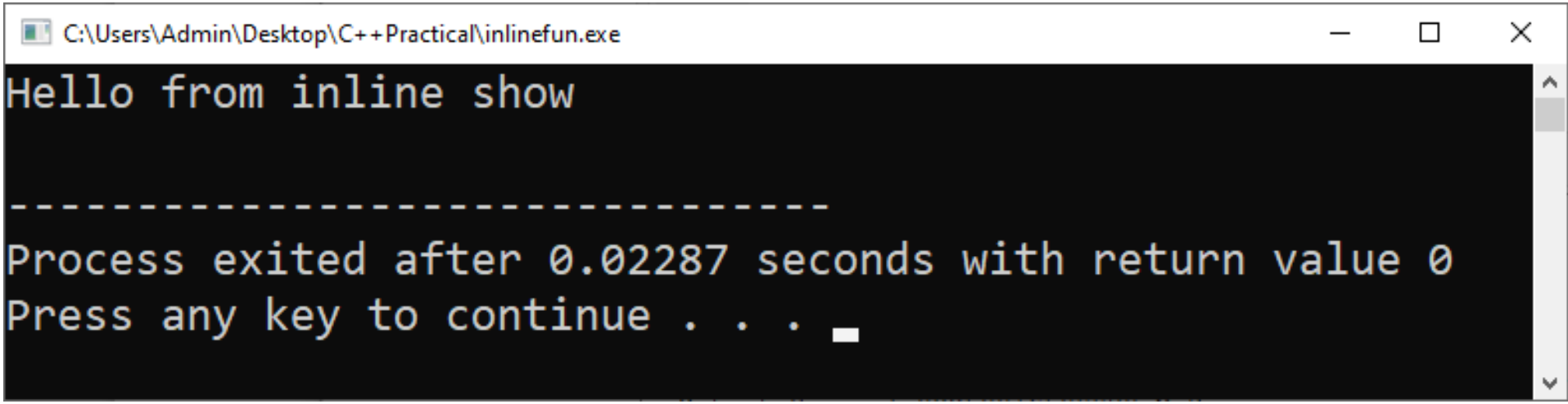
- **Some Important Points About Inline Function**
- Inline function **executes faster** than normal function.
- All inline functions **must be defined prior to their use.**
- Inline function **does not work** as inline when **code contains loops, recursions, goto, switch, static variable** etc.
- They are used where function definition are small so calling cost and overhead for normal functions can be minimized.

- E.g.

inlinefun.cpp

```
1   #include <iostream>
2   using namespace std;
3   inline void show( );
4
5   int main( )
6   {
7       show( );
8       return 0;
9   }
10
11  inline void show( )
12  {
13      cout<<"Hello from inline show\n";
14  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\inlinefun.exe
Hello from inline show

Process exited after 0.02287 seconds with return value 0
Press any key to continue . . .

Function Overloading

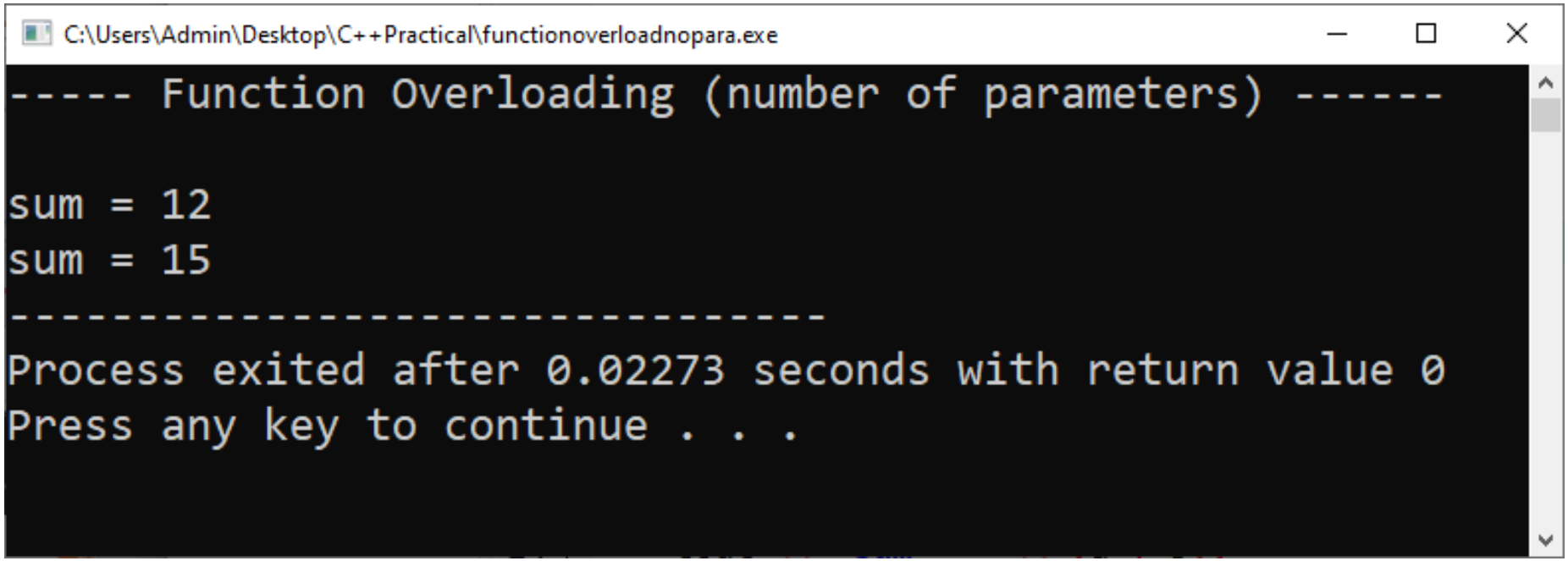
- **Overloading** refers to the **use of the same thing** for **different purpose**
- **Function overloading** means the **same function name** can be **used** to **create multiple functions** that **perform** a variety of **different tasks**.
- It is used to enhance the readability of the program.
- Function overloading is also known as function polymorphism.
- In function overloading we can create number of functions with the same name but either number of arguments or type of arguments must be different.
- **Ways to overload a function**
  - By changing number of Parameters (arguments)
  - By having different types of Parameters (arguments)

- **Overloading functions that differ in terms of number of parameters**

functionoverloadnopara.cpp

```cpp
#include <iostream>
using namespace std;
void add(int a, int b);
void add(int a, int b, int c);
int main()
{
    cout<<"----- Function Overloading (number of parameters) ------"<<endl<<endl;
    add(10, 2);
    add(5, 6, 4);
    return 0;
}
void add(int a, int b)
{
    cout << "sum = " << (a + b);
}
void add(int a, int b, int c)
{
    cout << endl << "sum = " << (a + b + c);
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\functionoverloadnopara.exe
----- Function Overloading (number of parameters) ------
sum = 12
sum = 15

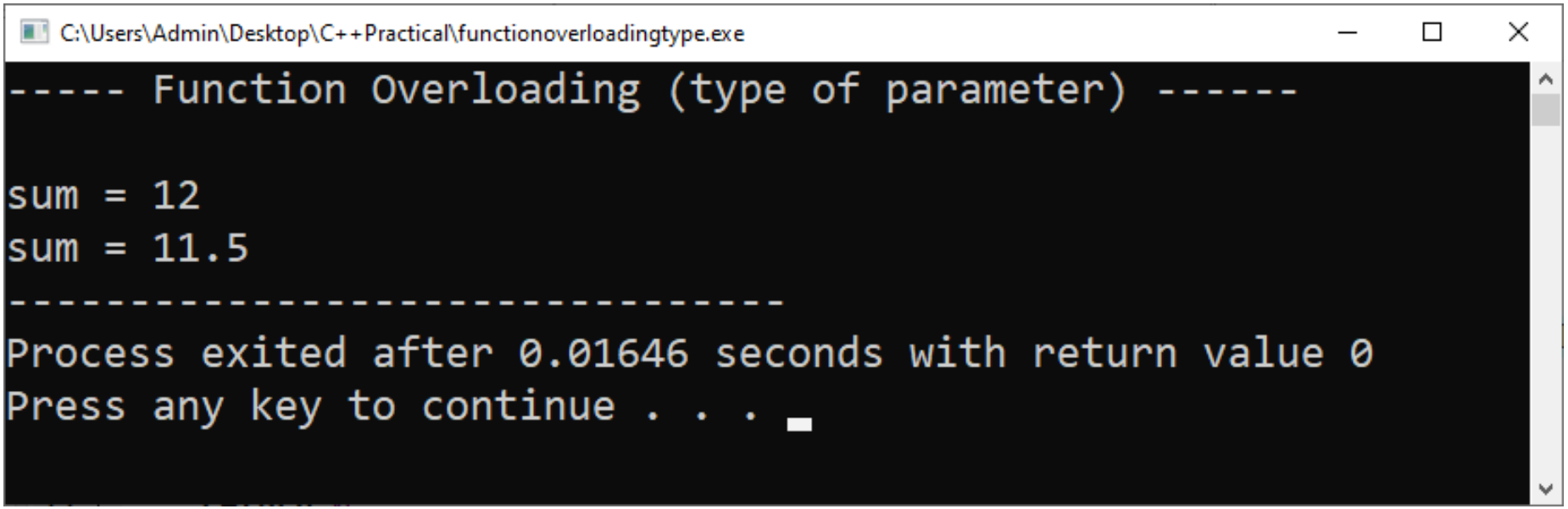
Process exited after 0.02273 seconds with return value 0
Press any key to continue . . .

- **Overloading functions that differ in terms of types of parameters**

functionoverloadingtype.cpp

```cpp
#include <iostream>
using namespace std;
void add(int a,int b);
void add(double a,double b);
int main()
{
    cout<<"----- Function Overloading (type of parameter) ------"<<endl<<endl;
    add(10, 2);
    add(5.3, 6.2);
    return 0;
}
void add(int a, int b)
{
    cout << "sum = " << (a + b);
}
void add(double a, double b)
{
    cout << endl << "sum = " << (a + b);
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\functionoverloadingtype.exe
----- Function Overloading (type of parameter) ------
sum = 12
sum = 11.5

Process exited after 0.01646 seconds with return value 0
Press any key to continue . . .

Function with Default Arguments

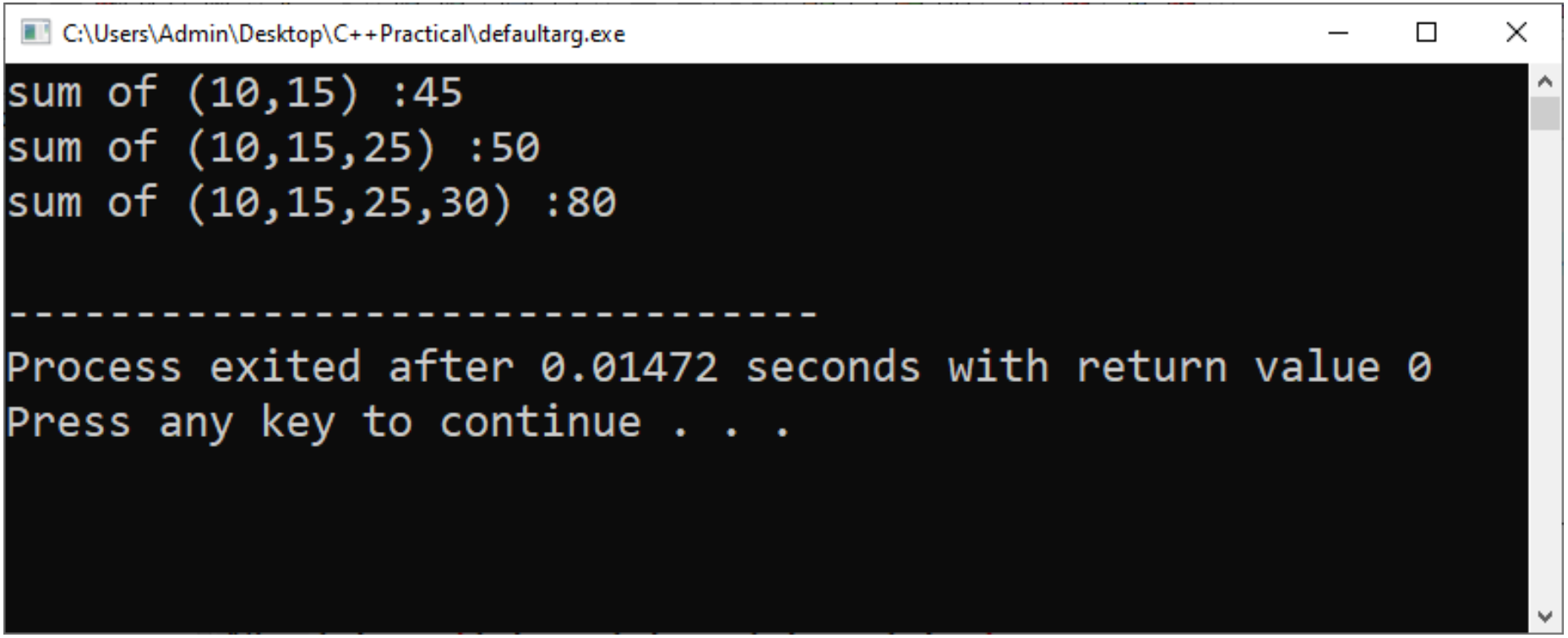
- Whenever a function is called, the calling function must be provide all the arguments specified in the function's declaration.
- If the calling function does not provide the required arguments, the compiler raises an error.
- However, **C++ allows a function to be called without specifying all of its arguments**. This **can be achieved** by **assigning a default value to the argument**.
- **Default value** is **specified in the function declaration** and is **used when** the **value of the argument is not passed while calling the function.**
- If **function call doesn't specify the an argument**, the **default value** is **passed** as an **argument to the function**. In case, a **function call specifies an argument**, the **default value is overridden** and the **specified value is passed to the function**.

- E.g.

defaultarg.cpp

```cpp
#include<iostream>
using namespace std;
int sum(int x, int y, int z = 20, int w = 0); //assigning default values to z,w as 20 and 0

int main()
{
    cout<<"sum of (10,15) :"<< sum(10, 15) << endl;
    cout<<"sum of (10,15,25) :"<< sum(10, 15, 25) << endl;
    cout<<"sum of (10,15,25,30) :"<< sum(10, 15, 25, 30) << endl;
    return 0;
}
int sum(int x, int y, int z, int w)
{
    return (x + y + z + w);
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\defaultarg.exe
sum of (10,15) :45
sum of (10,15,25) :50
sum of (10,15,25,30) :80

Process exited after 0.01472 seconds with return value 0
Press any key to continue . . .

- **Advantages of Default Arguments in C++**
- **Use of default arguments in C++ increases the capabilities and the reusability of the code of an existing function.** We can consider or ignore the default arguments depending on the number of values passed during the function call.
- The reusability of the same function, again and again, reduces the length of the program.
- **Disadvantages of Default Arguments in C++**
- The compiler needs more time to execute the program. It uses the extra time to replace the remaining arguments with their default values during the function call.

6. CLASS AND OBJECTS IN C++

Class

- The most remarkable feature of C + + is a class.
- Classes are **user-defined data types** and it behaves like built-in types of programming language.
- It is similar to a structure with the difference that it can also have functions besides data items.
- **Class** is a way to bind **data** and its associated **functions** together (A class can have data members as well as function members)
- Class **specification** has two parts:
  - **Class Declaration**
  - **Class Function Definitions**
- The Class declaration describes the type and scope of its members
- The Class function definition describe how the class functions are implemented

- The **declaration** begins with the keyword **class** followed by the name of the class.
- The body of class is enclosed within **braces { }** and terminated by **semicolon (;)**
- The variables and functions collectively called as **class members**.
- The **variables** declared inside the class are known as **data members** and the **function** known as **member function.**
- **Member functions** are also called **methods and services**

- Variable and function usually grouped under two sections i.e. private and public
- Keyword **private** and **public** are known as **visibility labels.** These are followed by a **colon (:)**
- The **data members** are usually declared as **private** and the **member functions** as **public**
- **If both keywords are missing then, by default, all members are private**. Such a class is completely hidden from the outside world and does not serve any purpose.
- **Only the member function can have access to the private data members and private functions.**
- **The public members can be accessed from outside the class.**

Access specifiers of a class

- **Private**
  - The data members declared as private can be accessed only from within the class
- **Public**
  - The data members and functions declared in the public section can be accessed by any function in the outside the class.
- **Protected**
  - Members of the class that are protected can only be accessed by the member function of the class, derived class and friend functions of the class.

Specifiers	Within Same Class	In Derived Class	Outside the Class
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

Rules for naming classes

- A class name **must begin with a letter** and can be followed by a sequence of letters(A-Z),digits(0-9) and underscore (_)
- **Special character** such as ? - + * / \ ! @ # $ % ^ ( ) [ ] { }, ; : . **Cannot be used** in the class name
- A class name **must not be the same as a reserved keyword** such as using , public etc.
- Class name must start with an uppercase letter(Although this is not mandatory). If class name is made of more than one word, then first letter of each word must be in uppercase.

  e.g. class Employee

The general form of class declaration

```
class class_name
{
private :
Variable declaration;
Function declaration;
public :
Variable declaration;
Function declaration;
protected :
Variable declaration;
Function declaration;
};
```

E.g.

```
class Employee
{
        char name[20];          //Data members
        int empid;

        public:
        void getdata()          //Member function
        {
                ..........
        }
};
```

Objects

- Class variables are known as objects.
- Defining an object is similar to defining a variable of any data type.
- Objects are instances of class , which holds the data variables declared in class and the member functions work on these class objects.
- Syntax

class_name object_name;

e.g.

Employee e1;

Accessing class members

- Syntax:

  object-name . function-name(argument list);

  (The dot operator “.” is called as class member access operator)

- e.g.

  x.getdata(200,52.6);

  x.putdata( );

Defining Member Function

- A **member function** of a class is a **function that has its definition or its prototype within the class definition like any other variable.**
- A member function, is a function declared as a member of a class
- A member function performs an operation required by the class
- **Members functions are usually put in the public part of the class because they have to be called outside the class either in program or in a function**
- Member functions can be defined into two parts:
  - **Inside the class definition**
  - **Outside the class definition**

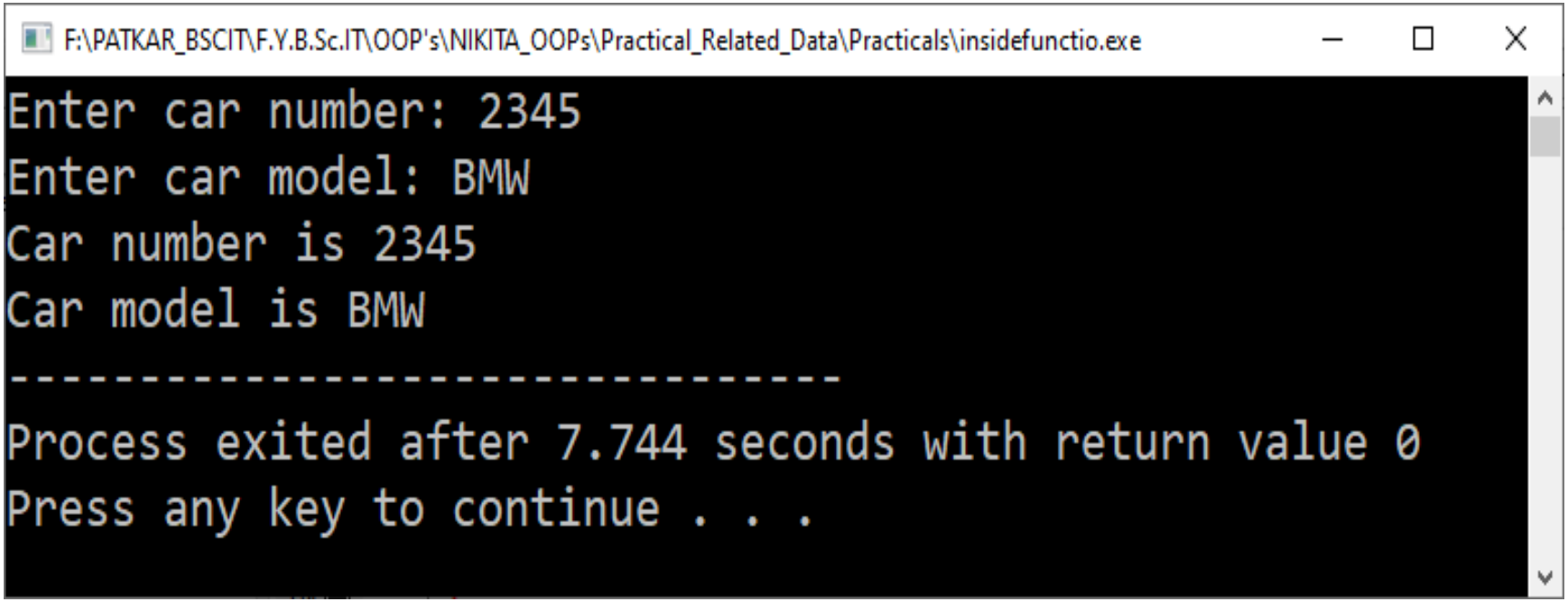
- **Inside the class definition**
- When the member functions of a class are defined **inside the class** itself they are called as **internally defined member functions.**

e.g.

insidefunctio.cpp

```cpp
#include <iostream>
using namespace std;
class car        //The class name is car
{
  private:
    int car_number;       //data members which is private
    char car_model[10];   //data members which is private
  public:
    void getdata()        //Definition of function inside the class
     {
       cout<<"Enter car number: ";
       cin>>car_number;
       cout<<"Enter car model: ";
       cin>>car_model;
     }
    void showdata()       //Definition of function inside the class
     {
       cout<<"Car number is "<<car_number<<endl;
       cout<<"Car model is "<<car_model;
     }
};
                          // main function starts
int main()
 {
    car c1;           //In main function object created c1 is object of class car
    c1.getdata();     //getdata()  function get called using object
    c1.showdata();    //showdata() function get called using object
    return 0;
 }
```

Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\insidefunctio.exe
Enter car number: 2345
Enter car model: BMW
Car number is 2345
Car model is BMW

Process exited after 7.744 seconds with return value 0
Press any key to continue . . .

- **Outside the class definition**
- Here the functions are **defined outside** the class however they are **declared inside the class**.
- When the member functions of a class are defined **outside the class** they are called as **externally defined member functions**.
- The general form of definition is

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

- Symbol **: :** is called **scope resolution operator**

Syntax :

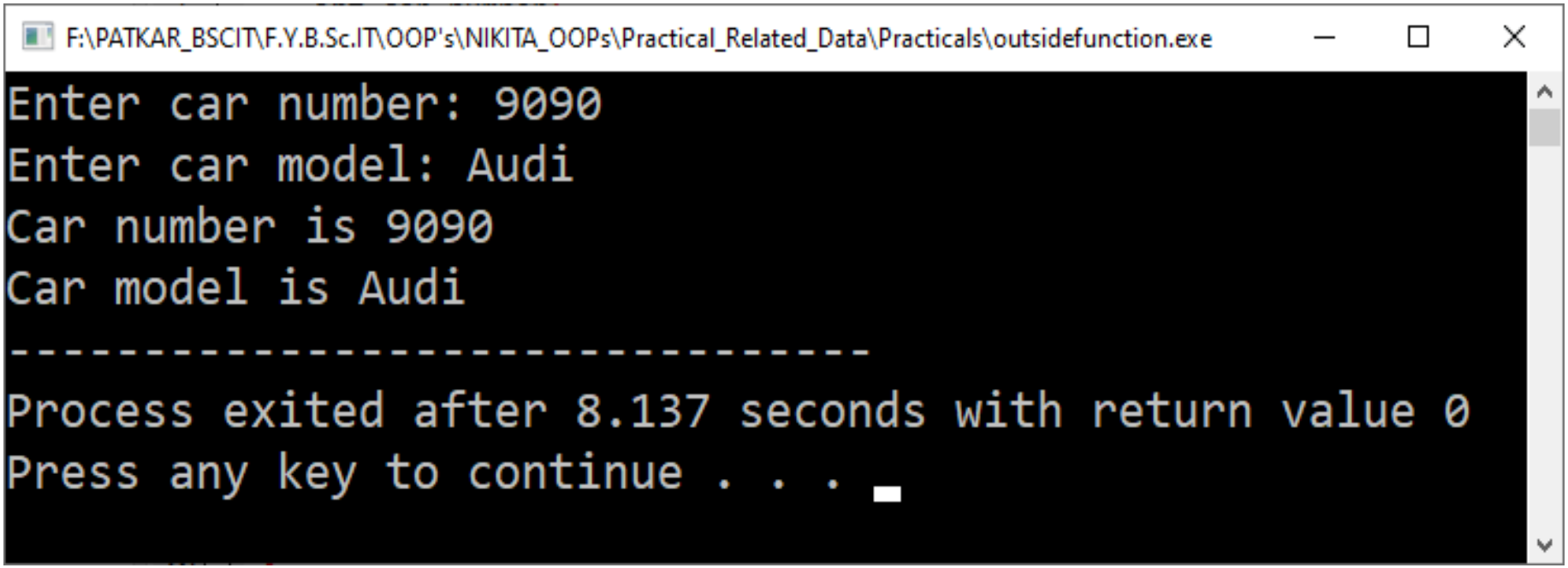
```
class  class_name
{ …….. ……..
   public:
   return-type   function-name(argument); //function declaration
};
//function definition outside class
return-type   class-name :: function-name (argument)
{
          ……….;                    // function definition
}
```

e.g.

outsidefunction.cpp

```cpp
#include<iostream>
using namespace std;
class Car
{
  private:
    int car_number;
    char car_model[10];
  public:
    void getdata();                          //function declaration
    void showdata();
};
void Car::getdata()                          // function definition
 {
   cout<<"Enter car number: ";
   cin>>car_number;
   cout<<"Enter car model: ";
   cin>>car_model;
 }
void Car::showdata()
 {
   cout<<"Car number is "<<car_number<<endl;
   cout<<"Car model is "<<car_model;
 }
// main function starts
int main()
 {
    Car c1;
    c1.getdata();
    c1.showdata();
    return 0;
 }
```

Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\outsidefunction.exe
Enter car number: 9090
Enter car model: Audi
Car number is 9090
Car model is Audi

Process exited after 8.137 seconds with return value 0
Press any key to continue . . .

Accessing Private Data

- The private members of a class are not accessible outside the class not even with the object name.
- However, they can be accessed indirectly through the public member functions of that class.

Project | Classes | Debug

privatedata.cpp

```
1   #include<iostream>
2   using namespace std;
3   class MyClass
4   {
5     int y;
6     void get()
7     {
8       cout<<"Enter Value :";
9       cin>>y;
10    }
11    void shw()
12    {
13      cout<<"Value is :"<<y;
14    }
15  };
16  int main()
17  {
18    MyClass myobj;
19    myobj.get();
20    myobj.shw();
21    return 0;
22  }
```

Compiler (5) | Resources | Compile Log | Debug | Find Results | Close

Line	Col	File	Message
		C:\Users\Admin\Desktop\C++Practical\privatedata.cpp	**In function 'int main()':**
6	8	C:\Users\Admin\Desktop\C++Practical\privatedata.cpp	[Error] 'void MyClass::get()' is private
19	13	C:\Users\Admin\Desktop\C++Practical\privatedata.cpp	[Error] within this context
11	8	C:\Users\Admin\Desktop\C++Practical\privatedata.cpp	[Error] 'void MyClass::shw()' is private

Accessing private data outside its scope will give an error as above

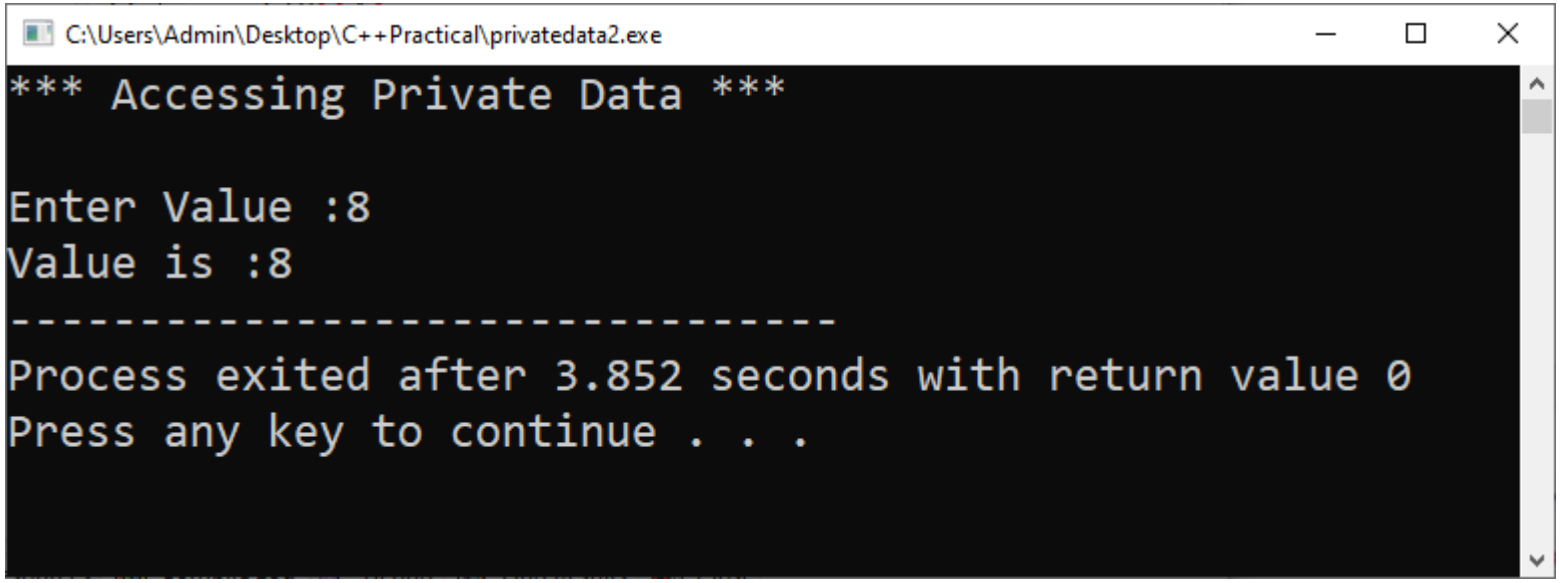
privatedata2.cpp

```
1   #include<iostream>
2   using namespace std;
3   class MyClass
4   {
5     int y;
6     public:
7     void get()
8     {
9       cout<<"*** Accessing Private Data ***\n\n";
10        cout<<"Enter Value :";
11      cin>>y;
12    }
13    void shw()
14    {
15      cout<<"Value is :"<<y;
16    }
17  };
18  int main()
19  {
20    MyClass myobj;
21    myobj.get();
22    myobj.shw();
23    return 0;
24  }
```

Compile Log | Debug | Find Results | Close

```
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\Admin\Desktop\C++Practical\privated
```

Output:


C:\Users\Admin\Desktop\C++Practical\privatedata2.exe
*** Accessing Private Data ***
Enter Value :8
Value is :8

Process exited after 3.852 seconds with return value 0
Press any key to continue . . .

Passing an object as an argument

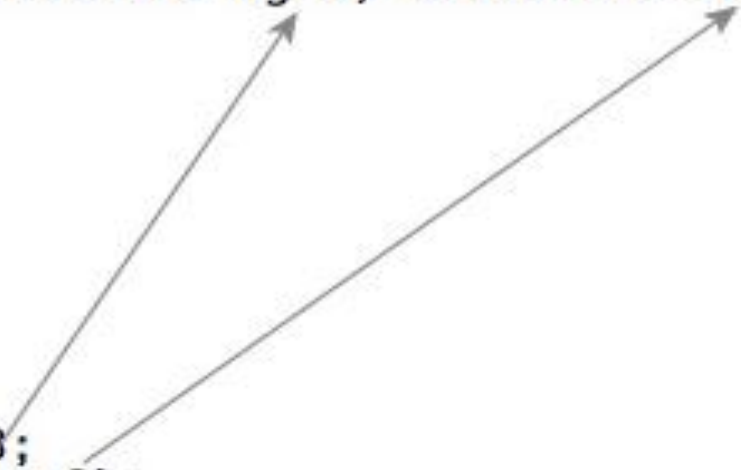
- To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables
- Syntax:

  function-name(object-name);

- General form of passing object as argument

How to pass objects to a function?

```
class className {
      ... .. ...

    public:
    void functionName(className agr1, className arg2)
    {
        ... .. ...
    }

    ... .. ..

};

int main() {

    className o1, o2, o3;
    o1.functionName (o2, o3);
}
```

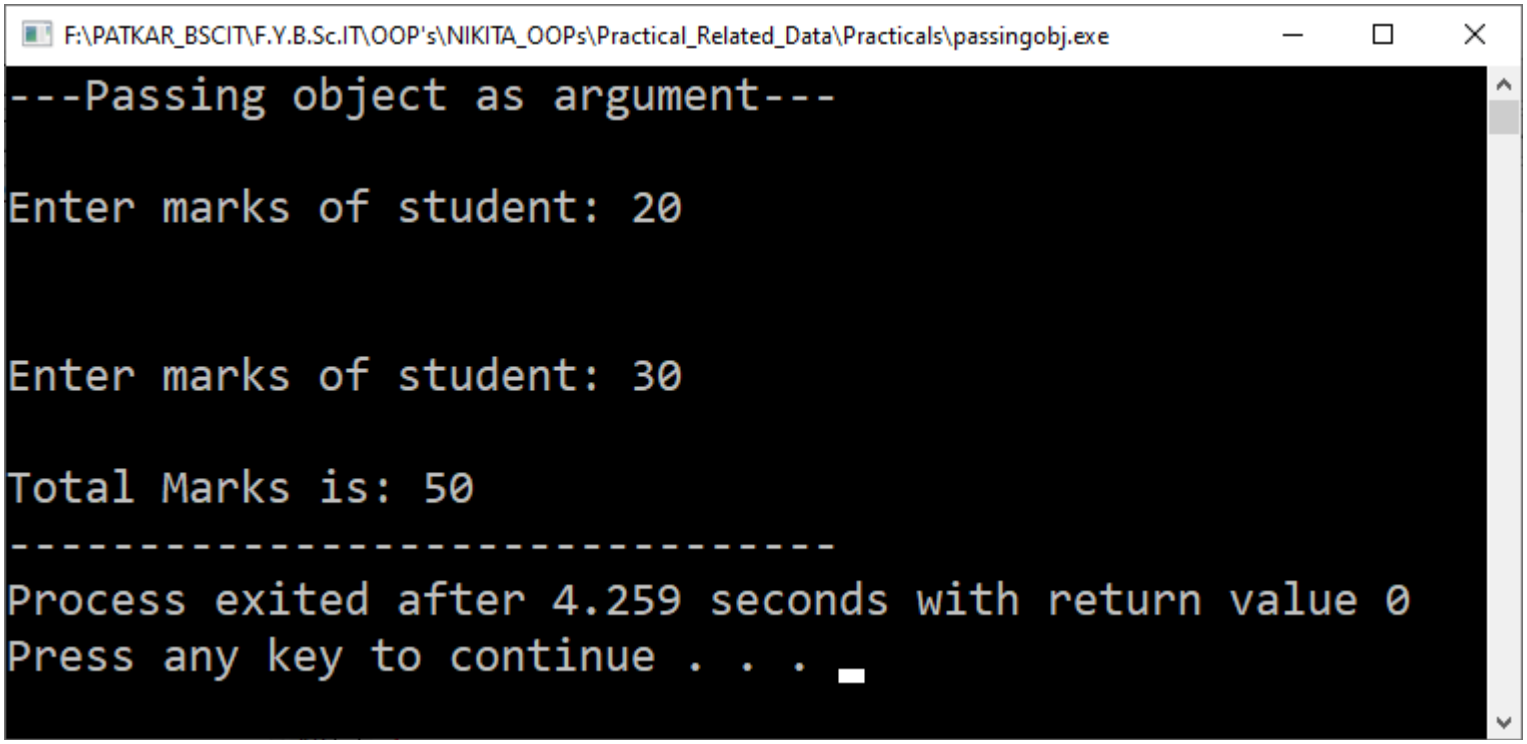


e.g.

passingobj.cpp

```
1  #include<iostream>
2  using namespace std;
3  class Student
4  {
5  int marks;
6  int totalmarks;
7  public:
8   void entermarks()
9   {
10      cout<<endl<<"\nEnter marks of student: ";
11      cin>>marks;
12  }
13  void addmarks(Student m1,Student m2)   //passing object as argument
14  {
15      totalmarks=m1.marks+m2.marks;
16  }
17  void displaymarks()
18  {
19      cout<<"\nTotal Marks is: "<<totalmarks;
20  }
21 };
22 int main()
23 {
24 cout<<"---Passing object as argument---";
25 Student s1,s2,s3;
26 s1.entermarks();
27 s2.entermarks();
28 s3.addmarks(s1,s2);
29 s3.displaymarks();
30 return 0;
31 }
```

Output :


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\passingobj.exe
---Passing object as argument---
Enter marks of student: 20
Enter marks of student: 30
Total Marks is: 50

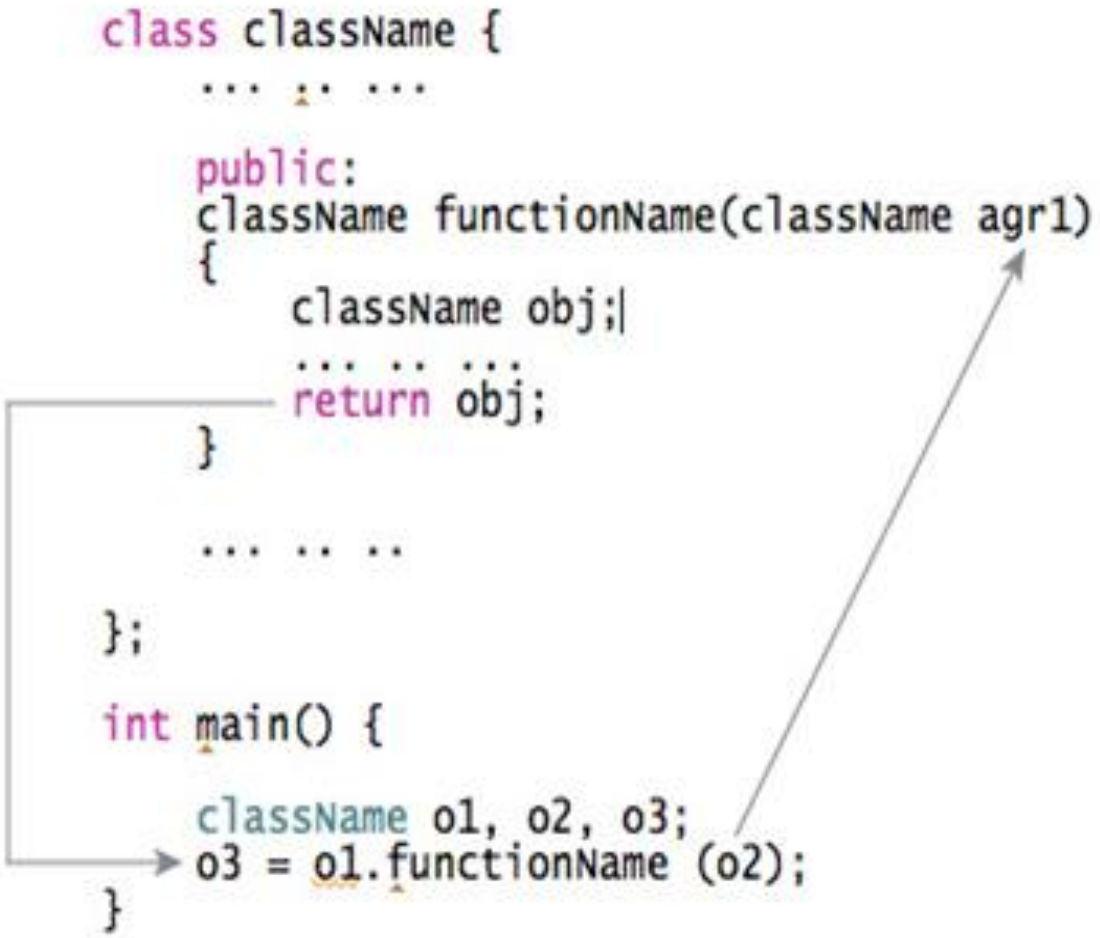
Process exited after 4.259 seconds with return value 0
Press any key to continue . . .

Returning object from function

- Just as it is possible for a function to return value it is also possible for a function to return an object

- General form of returning object from function

How to return an object from the function?

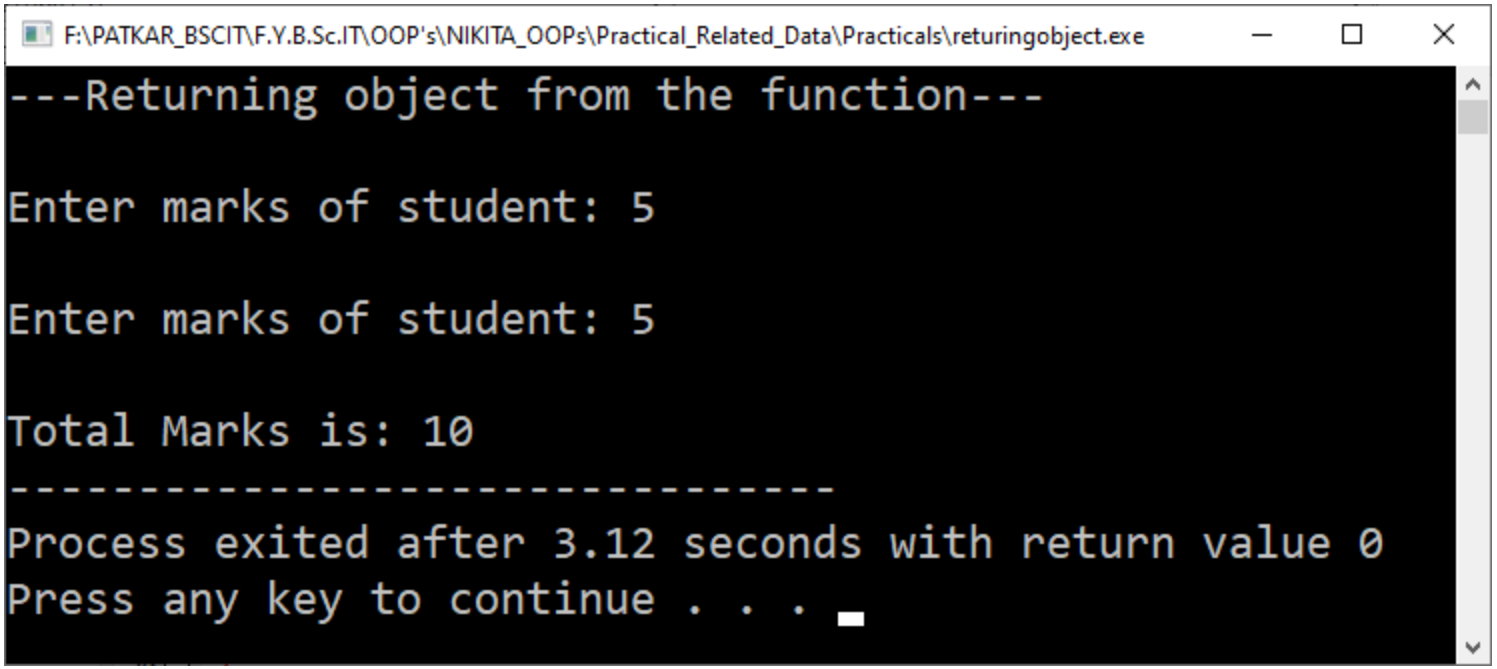


e.g.

returingobject.cpp

```cpp
#include<iostream>
using namespace std;
class Student
{
int marks;
int totalmarks;
public:
 void entermarks()
 {
    cout<<endl<<"Enter marks of student: ";
    cin>>marks;
 }
Student addmarks(Student m1)
 {
    Student m3;
    m3.totalmarks=marks+m1.marks;
    return m3;                          //returning object
 }
 void displaymarks()
 {
    cout<<"\nTotal Marks is: "<<totalmarks;
 }
};
int main()
{
cout<<"---Returning object from the function---"<<endl;
Student s1,s2,s3;
s1.entermarks();
s2.entermarks();

s3=s1.addmarks(s2);
s3.displaymarks();
return 0;
}
```

Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\returingobject.exe
---Returning object from the function---
Enter marks of student: 5
Enter marks of student: 5
Total Marks is: 10

Process exited after 3.12 seconds with return value 0
Press any key to continue . . .

Arrays of object

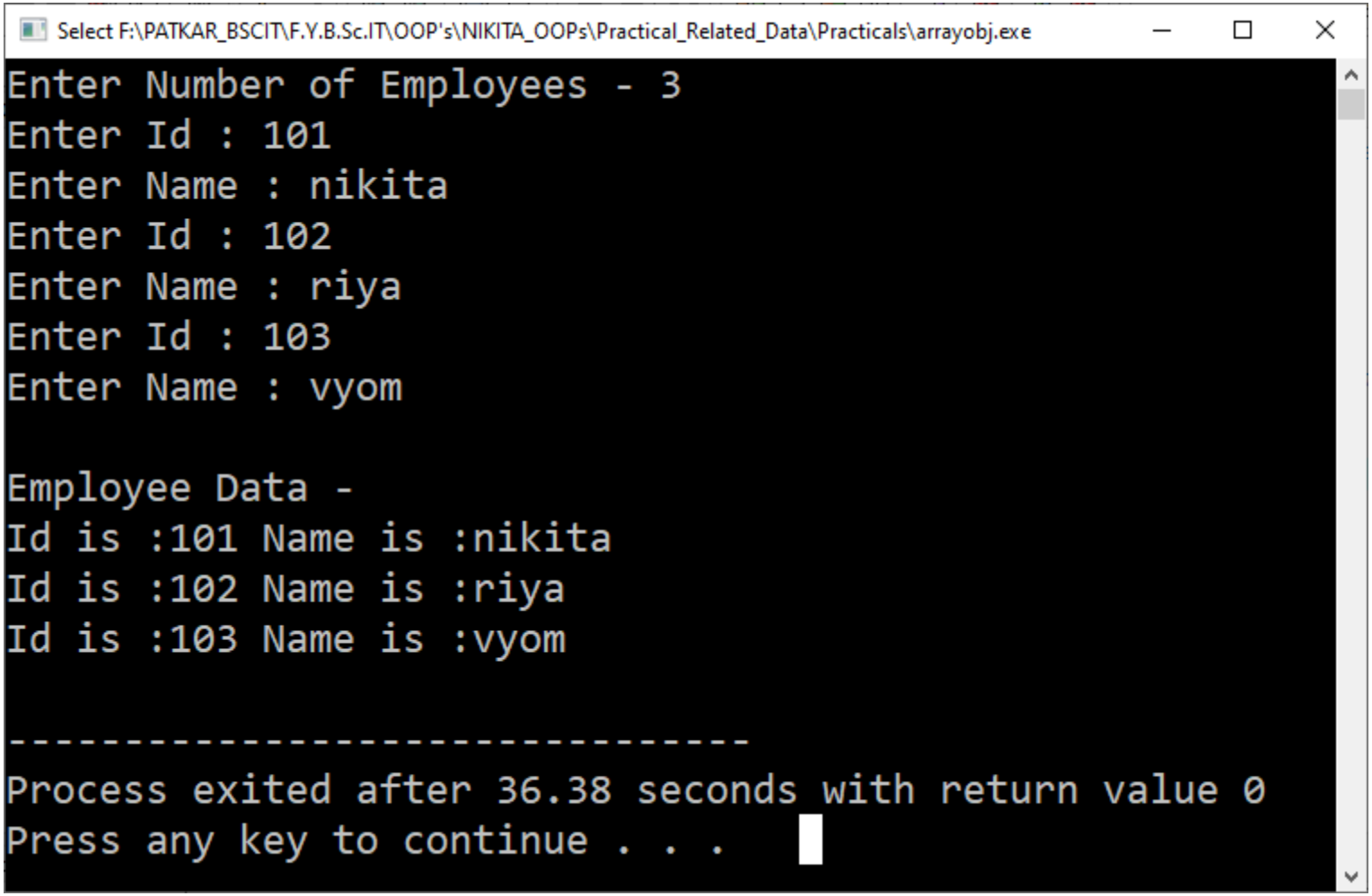
- Similar to array of any basic data types we can create array of object of any class
- The array of type class contains the objects of the class its individual elements. Thus , array of a class type is also known as an array of objects.
- **An array of objects is declared in the same way as an array of any built-in data type.**

E.g.

arrayobj.cpp

```
#include<iostream>
using namespace std;
class Employee
{
  int id;
  char name[30];
  public:
  void getdata();                          // Declaration of function
  void putdata();                          // Declaration of function
};
void Employee::getdata()                   // Defining the function outside the class
{
  cout<< "Enter Id : ";
  cin>>id;
  cout<< "Enter Name : ";
  cin>>name;
}
void Employee::putdata()                   // Defining the function outside the class
{
  cout <<"Id is :"<<id << " ";
  cout <<"Name is :"<<name << " ";
  cout << endl;
}
int main()
{
  Employee emp[30];                        // This is an array of objects having maximum limit of 30 Employees
  int n, i;
  cout << "Enter Number of Employees - ";
  cin >> n;

  for(i = 0; i < n; i++)
  {
    emp[i].getdata();                      // Accessing the function
  }
  cout <<"\nEmployee Data - " << endl;

  for(i = 0; i < n; i++)
  {                                        // Accessing the function
    emp[i].putdata();
  }
  return 0;
}
```

Output:


Select F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\arrayobj.exe
Enter Number of Employees - 3
Enter Id : 101
Enter Name : nikita
Enter Id : 102
Enter Name : riya
Enter Id : 103
Enter Name : vyom
Employee Data -
Id is :101 Name is :nikita
Id is :102 Name is :riya
Id is :103 Name is :vyom

Process exited after 36.38 seconds with return value 0
Press any key to continue . . .

Friend function

- Friend functions are **made to give private access to non-class functions.**
- Friend functions are actually not class member function
- A friend function of a class is **defined outside that class** scope **but it has the right to access all private and protected members of the class.**
- Friend functions are declared with the **friend keyword inside the class.**
- The function that are declared with the keyword friend are known as **friend function.**
- The **function definition** does not use either the keyword friend or the scope resolution operator (::)

- Syntax:

```
class class_name
{  … …
   public:
   … …
   friend return-type function-name(arguments);
};
```

Characteristics of friend function

- A friend function is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class , **it cannot be called using the object of that class**. It can **be invoked like a normal function** without the use of any object.
- Generally, **it has the objects as arguments**
- Unlike member functions, **it cannot access the members directly**. However, **it has to use the object and dot membership operator** with each member name to access both private and public members.
- A friend function can be declared in any of these sections(Private ,Public or protected)
- A friend function can be declared as a friend of more than one class

(before friend function....)

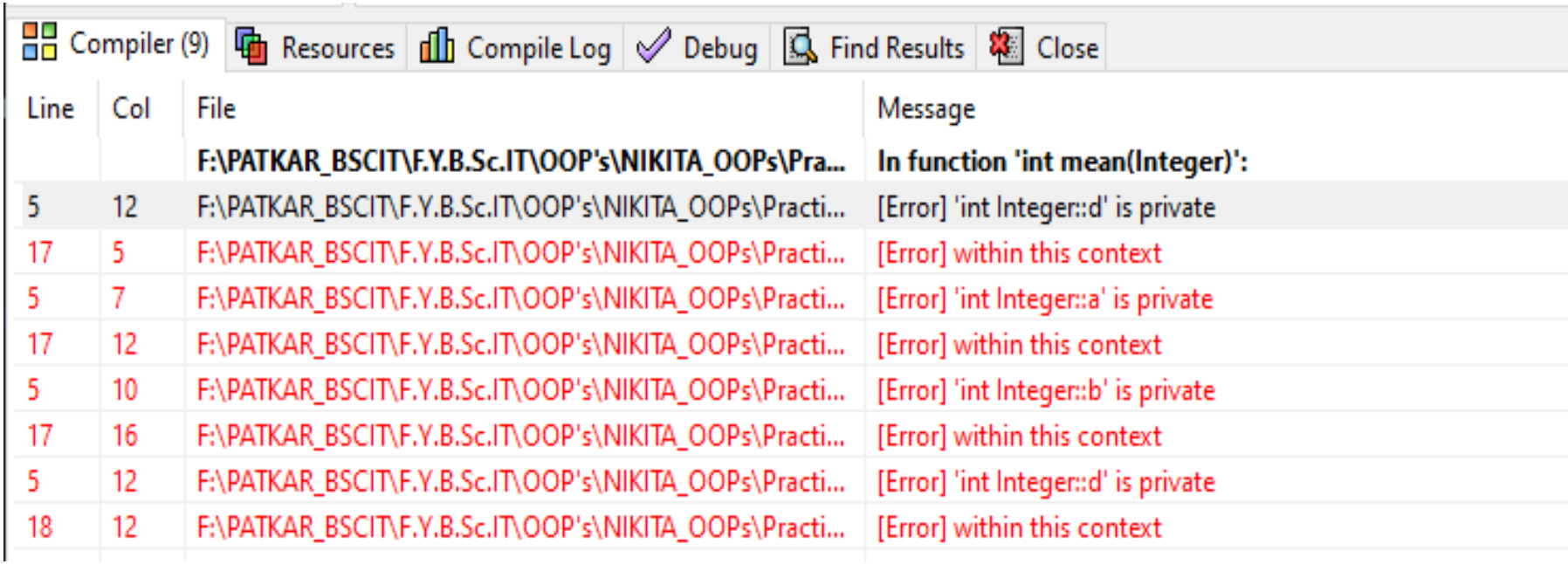
friendfun.cpp

```
1   #include<iostream>
2   using namespace std;
3   class Integer
4   {
5     int a, b,d;
6
7     public:
8       void set_value()
9       {
10      a=50;
11      b=30;
12      }
13  };
14
15  int mean(Integer s)
16  {
17    s.d = (s.a+s.b)/2;
18    return s.d;
19  }
20  int main()
21  {
22    Integer c;
23    c.set_value();
24    cout<< "Mean value:";
25    cout<<mean(c);
26    return 0;
27  }
```

v.imp question

(before friend function....)

It will give an Error while accessing private data outside the class



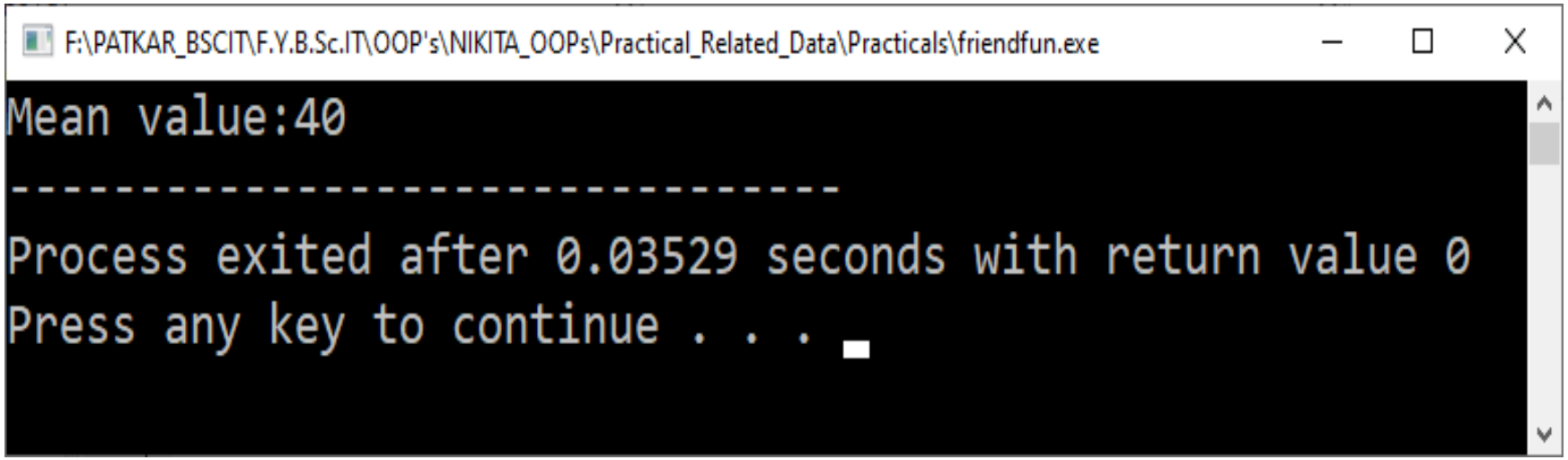
(Using friend function....)

e.g.

[*] friendfun.cpp

```
1  #include<iostream>
2  using namespace std;
3  class Integer
4  {
5    int a, b,d;
6
7    public:
8      void set_value()
9      {
10     a=50;
11     b=30;
12     }
13
14     friend int mean(Integer s);  //declaration of friend function
15 };
16
17 int mean(Integer s)   //friend function definition ."Integer"  is a class and "s" is a object of class
18 {
19   s.d = (s.a+s.b)/2;
20   return s.d;
21 }
22 int main()
23 {
24   Integer c;
25   c.set_value();
26   cout<< "Mean value:";
27   cout<<mean(c);          //calling friend function
28   return 0;
29 }
```

- Output


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\friendfun.exe
Mean value:40

Process exited after 0.03529 seconds with return value 0
Press any key to continue . . .

Static Data Members

- **Static data members** hold **global data** that is common to all the **objects** of the class.
- When the **member variables declaration** is **preceded** with **keyword static**, it tells the compiler that **one copy** of that **variable** will **exist** and **all the objects of** the **class** will **share that variable** (i.e. *All the objects of a class share the single copy of the static data member*)
- **Static data members** are **useful in situations** where either **a common item of information is to be shared among all objects OR the number of objects actually in existence is to be determined** (i.e. *To keep a track of how many objects being created in program*)

- It is **initialized** to **zero** when the first object of its class is created.
- Static variables are normally used to maintain **values common** to the **entire class**
- **Static data members is also called as class variable**
- **Static data members** can be **accessed using class name and scope resolution operator (::)**
- **Any data members** that **needs to be defined as static** is ***declared inside the class*** , but ***defined outside the class***
- Static variable must be **declared within a class (inside the class)**, with a **keyword static**
- **e.g.**

static int count;

- When the **static data member** is **defined outside the class** ,the **keyword static is not used**.
- **Syntax**

data-type class-name : : static-variable-name;

- **e.g.**

int abc :: count;

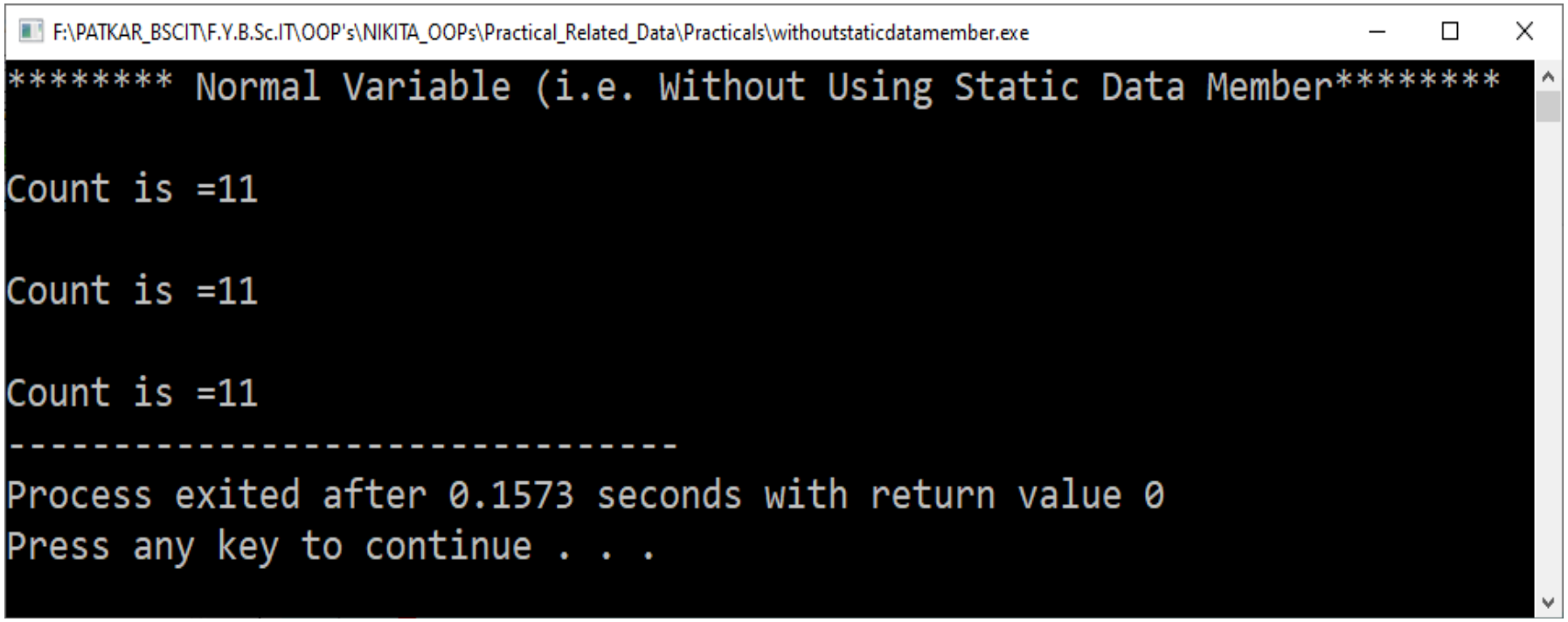
Using Normal Variable (Without Using Static Data Member)

- E.g.

withoutstaticdatamember.cpp

```
1   #include<iostream>
2   using namespace std;
3   class demo
4   {
5       int count;
6       public:
7           void get()
8           {
9               count=10;
10              count++;
11              cout<<"\n\nCount is ="<<count;
12          }
13  };
14
15  int main()
16  {
17      cout<<"******** Normal Variable (i.e. Without Using Static Data Member********";
18      demo d1,d2,d3;
19      d1.get();
20      d2.get();
21      d3.get();
22      return 0;
23  }
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\withoutstaticdatamember.exe
******** Normal Variable (i.e. Without Using Static Data Member********
Count is =11
Count is =11
Count is =11

Process exited after 0.1573 seconds with return value 0
Press any key to continue . . .

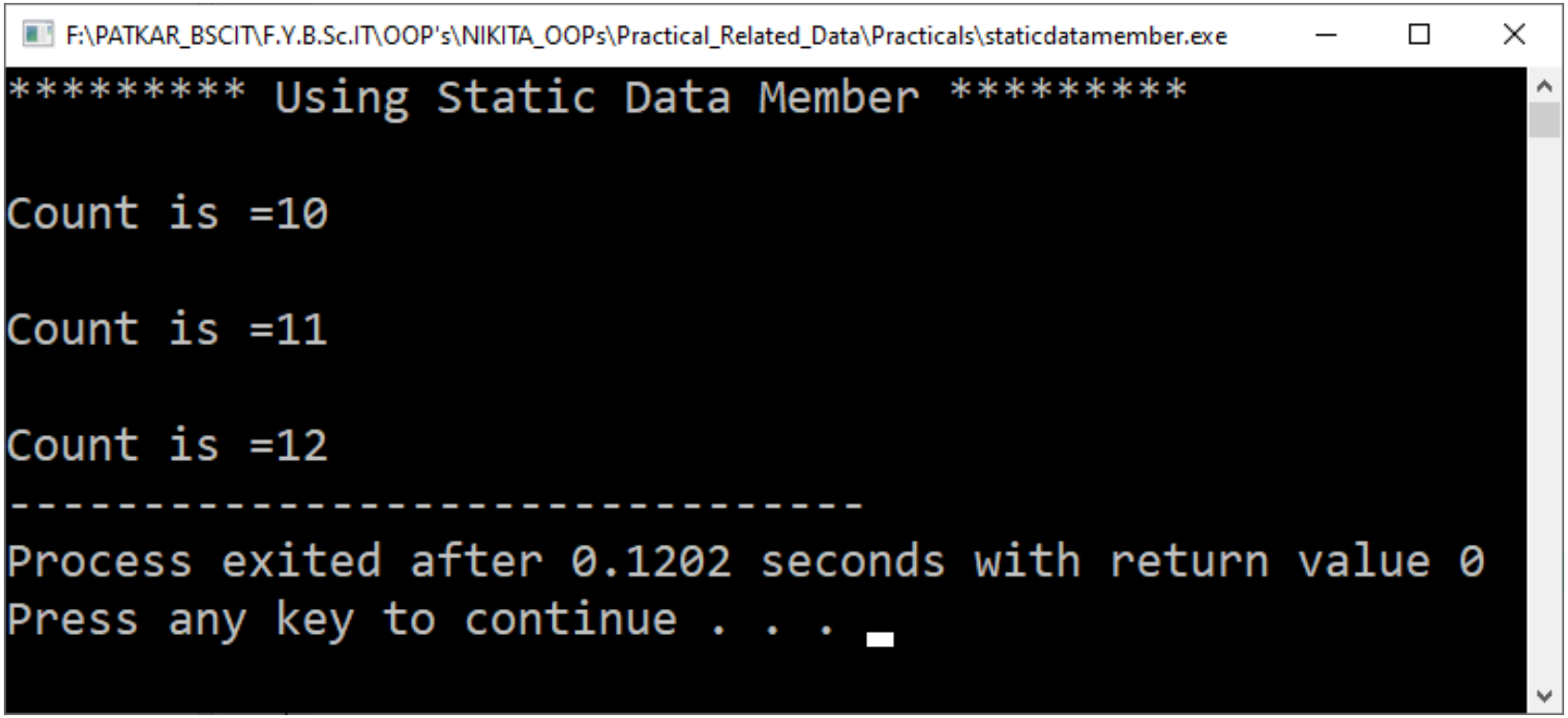
Using Static Data Member

- E.g.

staticdatamember.cpp

```
1   #include<iostream>
2   using namespace std;
3   class demo
4   {
5       static int count;          //declaration of static variable
6       public:
7           void get()
8           {
9
10              cout<<"\n\nCount is ="<<count++;
11
12          }
13  };
14  int demo::count=10;          //definition of static vriable
15  int main()
16  {
17      cout<<"********* Using Static Data Member *********";
18      demo d1,d2,d3;
19      d1.get();
20      d2.get();
21      d3.get();
22      return 0;
23  }
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\staticdatamember.exe
********** Using Static Data Member **********
Count is =10
Count is =11
Count is =12

Process exited after 0.1202 seconds with return value 0
Press any key to continue . . .

Static Member Function

- Like **static member variable**, **there is also** a **static member functions**
- A **static function** can **have access** to **only** other ***static data members and static member function declared in the same class***.
- **Static member function defined** by **prefixing** the **keyword static** to their **definition inside the class**
- A **static member function** can be **called** using the **class name** (instead of its objects) and **scope resolution operator (::)** as follows:
- Syntax

class-name : : function-name;

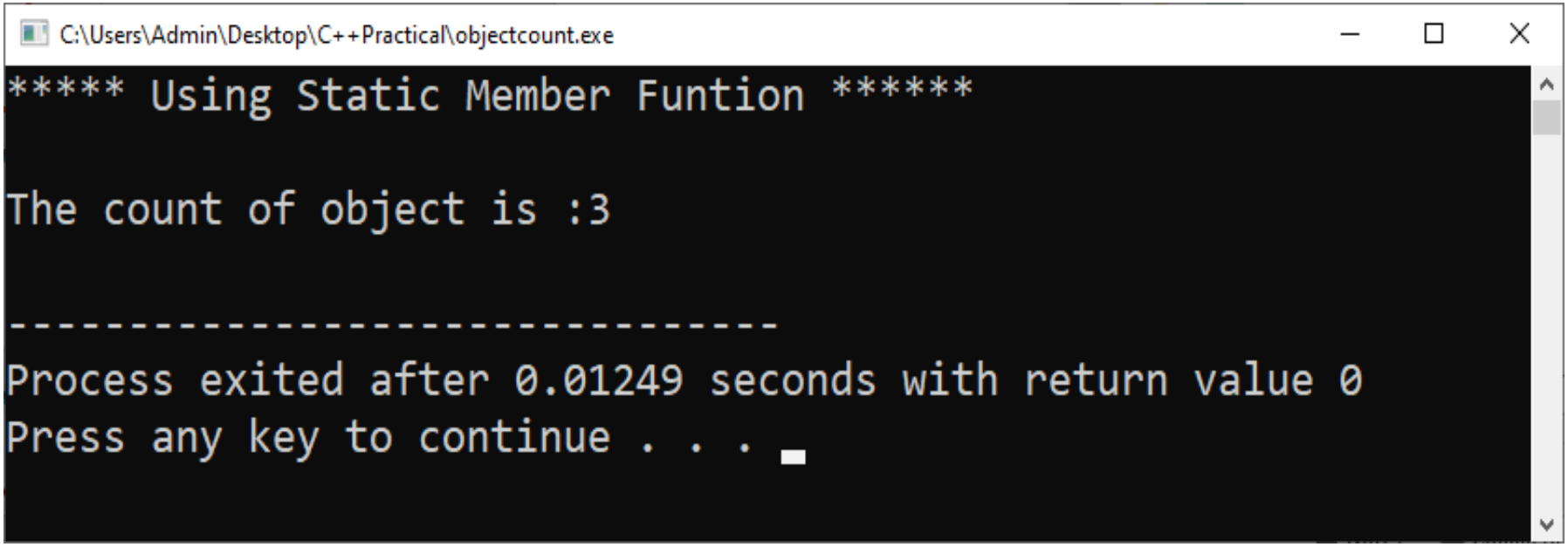
Using Static Member Function

- E.g.

objectcount.cpp

```
1   #include<iostream>
2   using namespace std;
3   class demo
4   {
5       static int count; //declaration of static variable
6       public:
7           void get()
8           {
9               count++;
10          }
11          static int show()  //deginition of static function
12          {
13              return count;
14          }
15  };
16  int demo::count;
17  int main()
18  {
19      cout<<"***** Using Static Member Funtion ******";
20      demo d1,d2,d3;
21      d1.get();
22      d2.get();
23      d3.get();
24      cout<<"\n\nThe count of object is :"<<demo::show()<<endl;
25      return 0;
26  }
```

- Output:



NOTE : Static data members use to keep a track of how many objects has been created

Constant Member Function

- A **'const' or a constant member function** can **only read or retrieve the data members** of the calling object **without modifying them.**
- If such member function **attempts** to **modify any data member** of the calling object, a **compile –time error is generated**.

- E.g.

Project | Classes | Debug

constfun.cpp

```
1  #include <iostream>
2  using namespace std;
3  class MyClass
4  {
5      int n;
6  public:
7      void getdata()
8      {
9          cout<<"Enter number : ";
10         cin>>n;
11     }
12     void show() const
13     {
14         n++;
15         cout<<"After Increment Value Is : "<<n;
16     }
17 };
18 int main(void)
19 {
20     MyClass obj;
21     obj.getdata();
22     obj.show();
23     return 0;
24 }
```

Compiler (2) | Resources | Compile Log | Debug | Find Results | Close

Line	Col	File	Message
		C:\Users\Admin\Desktop\C++Practical\constfun.cpp	In member function 'void MyClass::show() const':
14	4	C:\Users\Admin\Desktop\C++Practical\constfun.cpp	[Error] increment of member 'MyClass::n' in read-only object

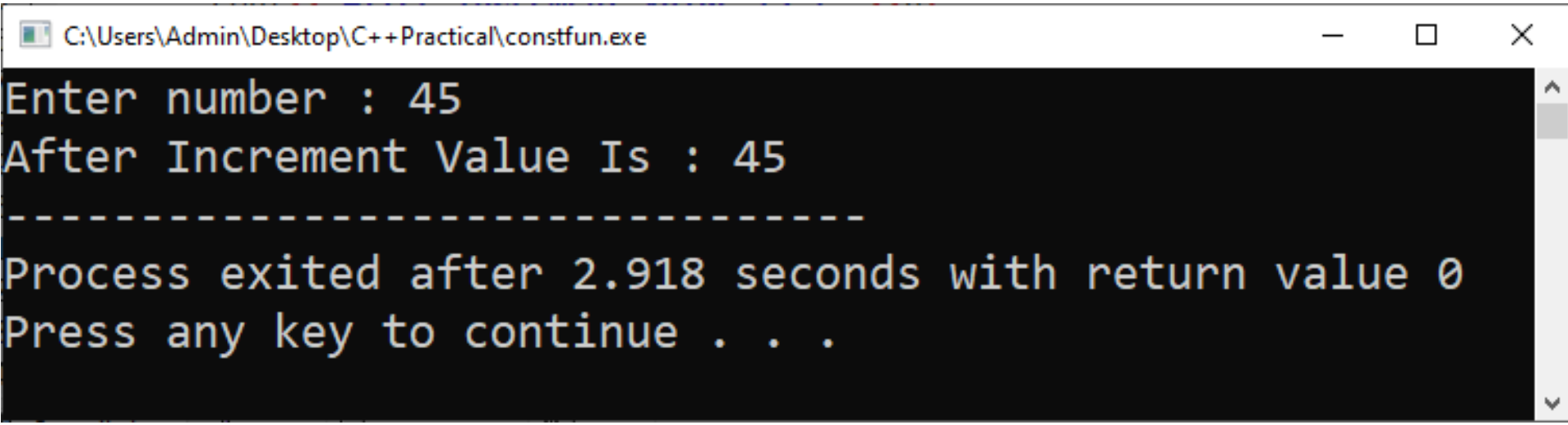
Line: 14 Col: 10 Sel: 0 Lines: 24 Length: 323 Insert Done parsing in 0 seconds

- E.g.

constfun.cpp

```
1   #include <iostream>
2   using namespace std;
3   class MyClass
4   {
5       int n;
6   public:
7       void getdata()
8       {
9           cout<<"Enter number : ";
10          cin>>n;
11      }
12      void show() const
13      {
14          //n++;
15          cout<<"After Increment Value Is : "<<n;
16      }
17  };
18  int main(void)
19  {
20      MyClass obj;
21      obj.getdata();
22      obj.show();
23      return 0;
24  }
```

- Output:


C:\Users\Admin\Desktop\C++Practical\constfun.exe
Enter number : 45
After Increment Value Is : 45

Process exited after 2.918 seconds with return value 0
Press any key to continue . . .

7. WORKING WITH CONSTRUCTOR AND DESTRUCTOR

Constructor

- Automatic initialization is carried out using a **special member function** called a constructor.
- A constructor is a **member function** that is **executed automatically** whenever an object is created.
- A constructor is a ‘special’ member function whose task is to initialize the objects of its class.
- It is special because its **name is same as the class**
- **Constructor is invoked** whenever an **object** of its associated class **is created**.
- It is called constructor because Constructors are used to construct the object of the class.
- It can be defined inside the class and outside the class as well using the scope resolution operator (**::**)

Difference between function and constructor

- **Constructor does not have any return type** like function
- The **name of the constructor must be of the same** of name as **that of the class**
- Whenever **object is created** then a **constructor is invoked**
- Function can be virtual function but it is not so incase of constructor

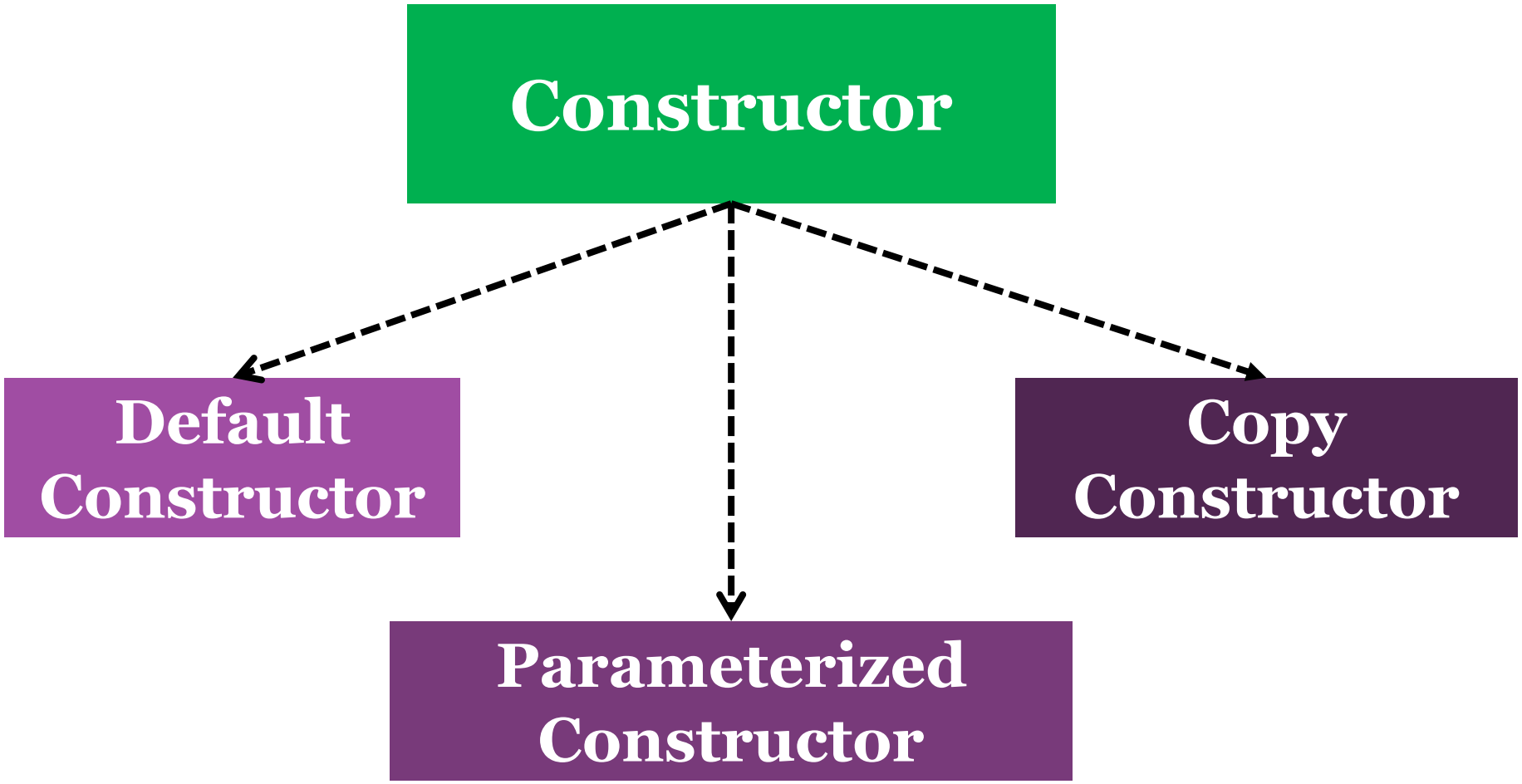
Characteristics of constructor

- The **name of constructor** function **should be the same** as the **name of the class**
- They should be **declared** in the **public** section.
- They are invoked (called) **automatically** when the **objects are created**.
- They **do not** have **return** types**, not even void** and they **cannot return values.**
- Constructors **cannot be inherited**, but they can be **called from the constructors of derived class**.
- Like other C++ functions, Constructors can have **default** arguments.
- Constructors cannot be virtual

- Syntax :

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

   abc()       //constructor
   {
        …….
   }
};
```

Types of Constructor



Default Constructor

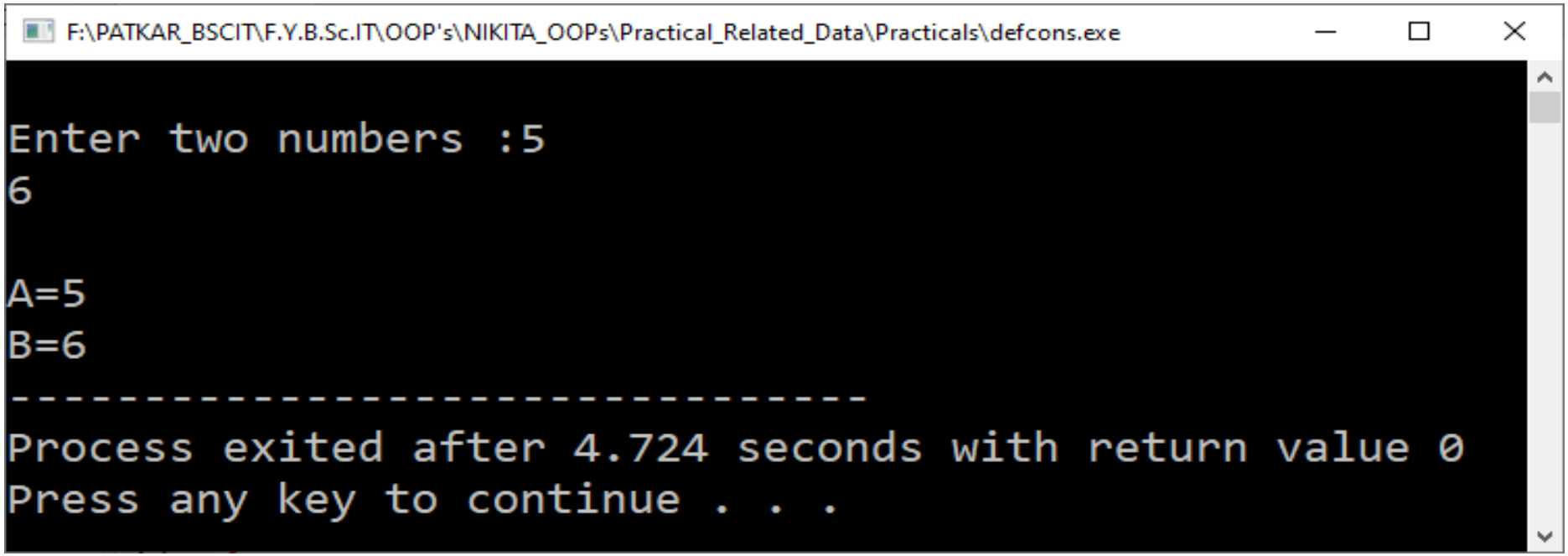
- If a constructor does not have any parameter list i.e. it cannot accept any parameters then it is called as "**default constructor**"(i.e. A constructor that accepts no parameters is called as default constructor)
- The **default constructor** is always **parameter less**.

e.g.

defcons.cpp

```cpp
#include <iostream>
using namespace std;
class construct
{
int a, b;
public:
     // Default Constructor
    construct( )
    {
        cout<<"\nEnter two numbers :";
        cin>>a>>b;
        cout<<"\nA="<<a;
        cout<<"\nB="<<b;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    return 0;
}
```

Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\defcons.exe
Enter two numbers :5
6
A=5
B=6

Process exited after 4.724 seconds with return value 0
Press any key to continue . . .

Parameterized Constructor

- It is also possible to create constructor with arguments and such constructors are called as **parameterized constructors** or **constructor with arguments**(i.e. The constructor that can take parameters or arguments are called as parameterized constructor)
- Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument
- For such constructors, it is **necessary** to **pass values** to the **constructor** when **object is created**
- This can be done by two ways:
  - By calling the constructor explicitly.
  - By calling the constructor implicitly.

- In implicit call we simply write the class name and then object name (in case of default constructor) or pass parameters but does not use the constructor name is any manner.
- In explicit call to constructor we explicitly call the constructor by writing its name and passing argument if any.
- The following declaration illustrates above method:

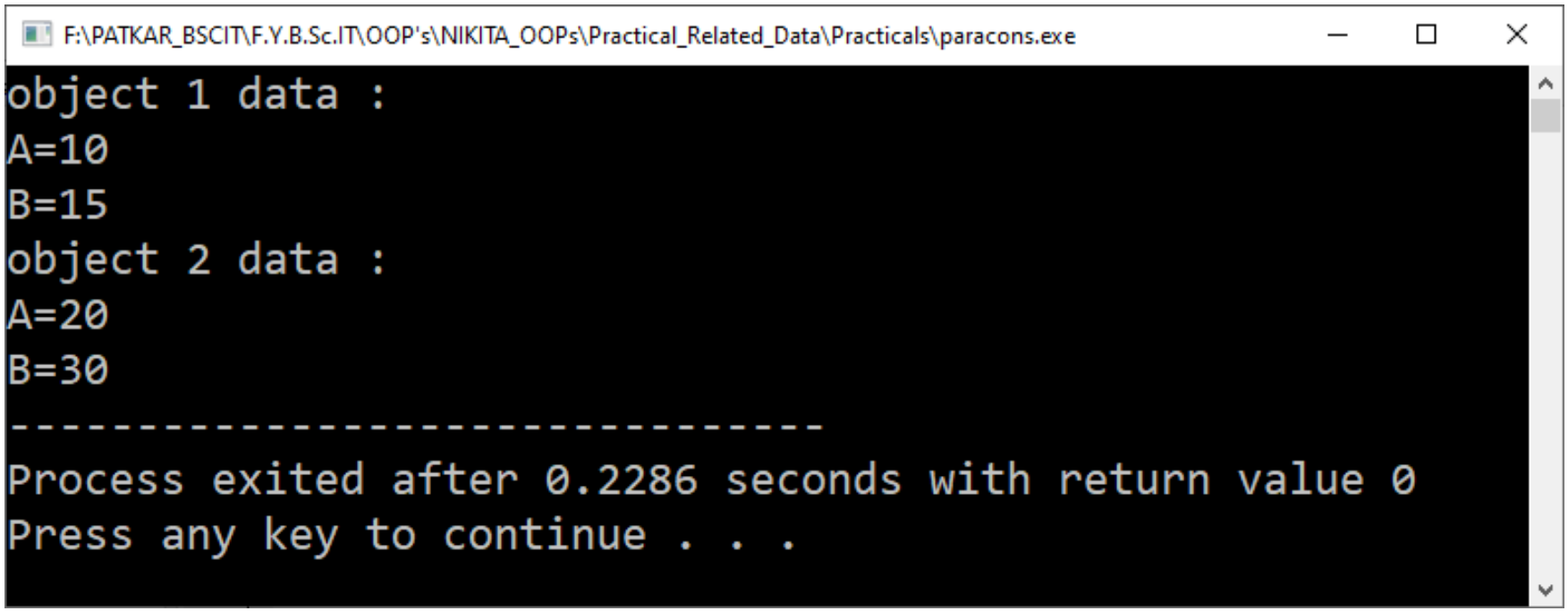
```
area  a = area (5, 6);  // explicit call
area  a (5, 6);  // implicit call
```

E.g.

paracons.cpp

```
1   #include<iostream>
2   using namespace std;
3   class Cube
4   {
5       int a,b;
6       public:
7           Cube(int n,int m)
8           {
9               a=n;
10              b=m;
11          }
12          void show()
13          {
14              cout<<"\nA="<<a;
15              cout<<"\nB="<<b;
16          }
17  };
18   int main()
19  {
20      Cube c1(10,15);          //Implicit calling
21      Cube c2=Cube(20,30);   //Explicit calling
22      cout<<"object 1 data :";
23      c1.show();
24      cout<<"\nobject 2 data :";
25      c2.show();
26      return 0;
27  }
```

Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\paracons.exe
object 1 data :
A=10
B=15
object 2 data :
A=20
B=30

Process exited after 0.2286 seconds with return value 0
Press any key to continue . . .

Copy Constructor

- A **copy constructor** is a constructor that **initializes a new object of a class** with the **values** of an **existing object of the same class**.
- It **creates a new object** ,which is **exact copy of the existing object** , hence it is called copy constructor.
- Syntax:

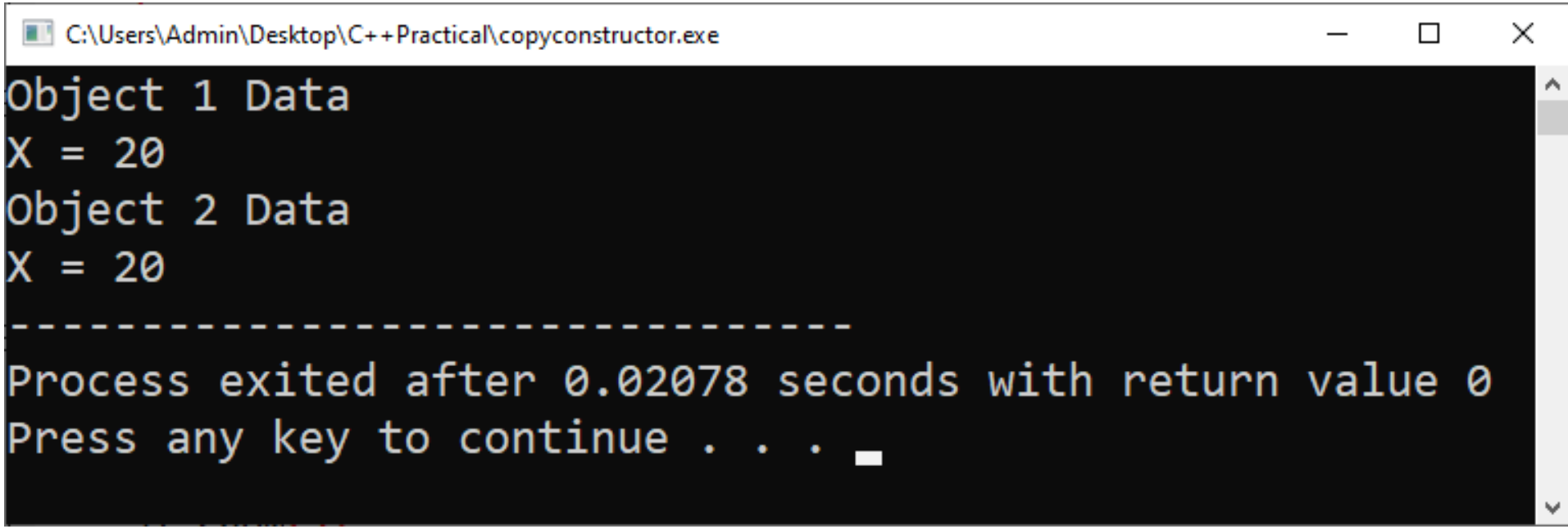
```
demo (demo &d)
{
        // copy constructor
}
```

- E.g.

copyconstructor.cpp

```cpp
#include <iostream>
using namespace std;
class A
{
   public:
    int x;
    A(int b)                  // parameterized constructor.
    {
      x=b;
    }
    A(A &i)                  // copy constructor
    {
        x = i.x;
    }
    void show()
    {
        cout<<"X = "<<x;
    }
};
int main()
{
    A a1(20);                 // Calling the parameterized constructor.
    A a2(a1);                 //  Calling the copy constructor.
    cout<<"Object 1 Data"<<endl;
    a1.show();
    cout<<"\nObject 2 Data"<<endl;
    a2.show();
    return 0;
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\copyconstructor.exe
Object 1 Data
X = 20
Object 2 Data
X = 20

Process exited after 0.02078 seconds with return value 0
Press any key to continue . . .

Dynamic Initialization of Objects

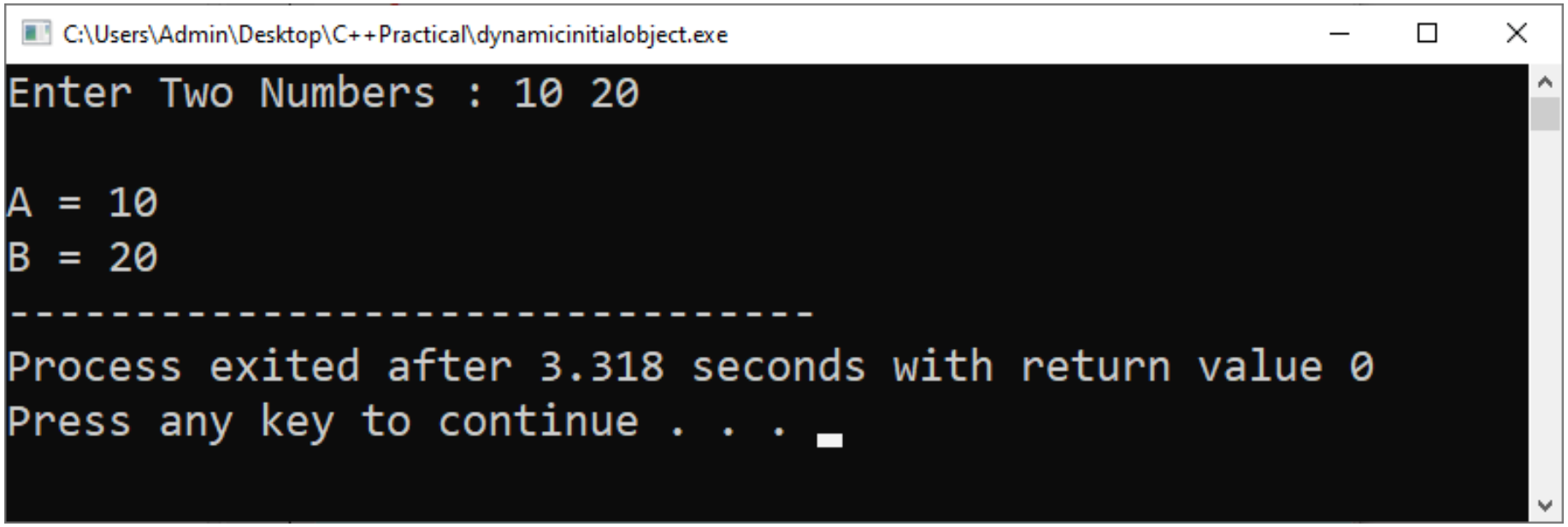
- Dynamic initialization of objects simply means **assigning the values to data members of class dynamically**. Here, the initial value of an object may be **provided during run time**.
- It can be achieved by using constructors and by passing parameters to the constructors.

- E.g.

dynamicinitialobject.cpp

```cpp
#include<iostream>
using namespace std;
class X
{
int a,b;
public:
    X(int p,int q)
    {
      a=p;
      b=q;
    }
    void disp()
    {
      cout<<"\nA = "<<a;
      cout<<"\nB = "<<b;
    }
};
int main()
{
    int a,b;
    cout<<"Enter Two Numbers : ";
    cin>>a>>b;
    X a1(a,b);                    // dynamic initialization
    a1.disp();
    return 0;
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\dynamicinitialobject.exe
Enter Two Numbers : 10 20
A = 10
B = 20

Process exited after 3.318 seconds with return value 0
Press any key to continue . . .

Dynamic Constructor

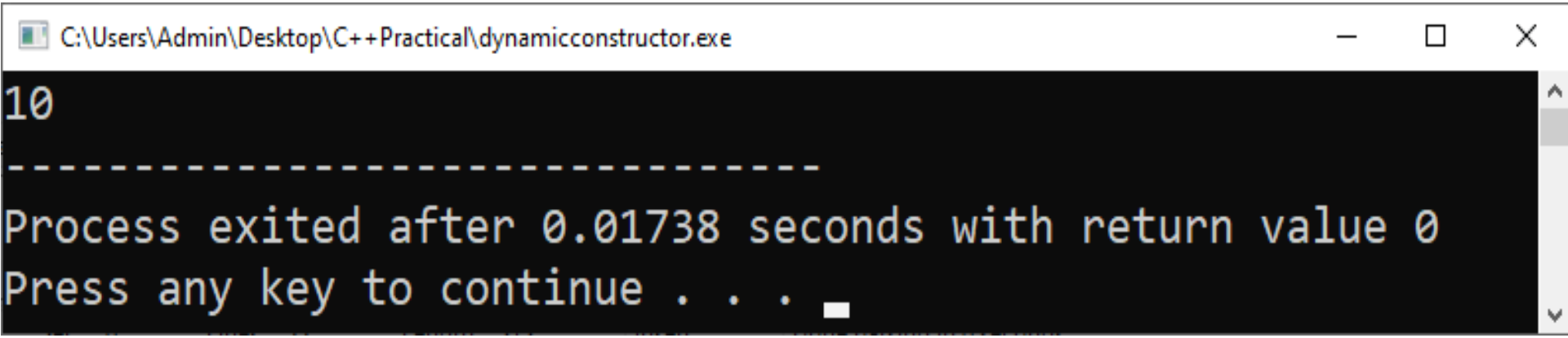
- When **in a constructor** we **create memory dynamically using dynamic memory allocator operator new**, then constructor is **known as dynamic constructor**.
- This is used to allocate(creating) memory while creating object.
- This will **enable the system** to **allocate the correct amount of memory** for **each object** when the objects are not of the same size, so resulting in the saving of memory.

- E.g.

dynamicconstructor.cpp

```cpp
#include<iostream>
using namespace std;
class dyn
{
    int *p;
    public:
      dyn()         // default constructor
      {
        p = new int;    // allocating memory at run time
        *p = 10;
      }
      void display()
      {
        cout<<(*p);
      }
};
int main()
{
      dyn obj1;
      obj1.display();
      return 0;
}
```

- Output:


C:\Users\Admin\Desktop\C++Practical\dynamicconstructor.exe
10

Process exited after 0.01738 seconds with return value 0
Press any key to continue . . .

Destructor

- A **destructor** is a special member function of a class that is executed (called) automatically whenever **an object of it's class goes out of scope**
- A destructor is a special member function that **works just opposite to constructor**, unlike constructor that are used for initializing an object, the **purpose of destructor** is to **destroy (or delete) the object** when it is no longer needed or goes out of scope.
- Destructors are **usually used to deallocate memory** and do other cleanup for a class object and its class members when the object is destroyed.

Characteristics of destructor

- A destructor will have exact **same name as the class prefixed with a tilde (~)**
- A **destructor** should be declared in the **public section** of the class
- A destructor **neither requires** any **argument** nor returns any value **(not even void)**
- A class can have only one destructor hence cannot be overloaded
- It is **automatically called** when object goes out of scope
- Destructor **releases memory space** occupied by the objects

- Syntax :

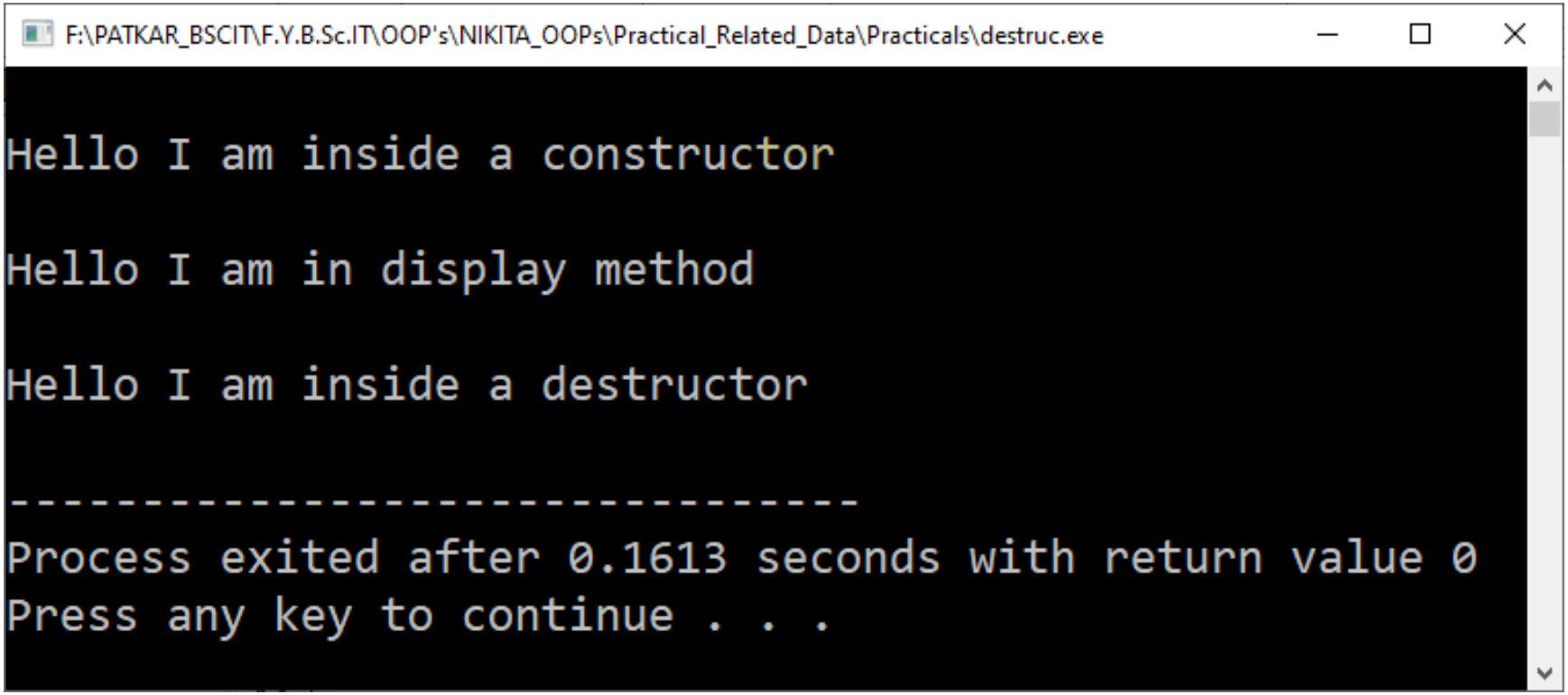
```
class class_name
  {
        public:
        class_name() //constructor
        {
                --------
         }

      ~class_name() //destructor
      {
          --------
      }
  };
```

- E.g.

destruc.cpp

```cpp
#include <iostream>
using namespace std;
class Example1          //class
{
    public:
    Example1()              // constructor
    {
        cout << "\nHello I am inside a constructor" << endl;
    }
    void display()
    {
        cout << "\nHello I am in display method" << endl;
    }

    ~Example1()   //destructor
    {
        cout << "\nHello I am inside a destructor" << endl;
    }
};
int main()
{
Example1 cc;        //object created
cc.display();       // display method called

/*....object cc goes out of scope ,now destructor is being called...*/
return 0;
}
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\destruc.exe
Hello I am inside a constructor
Hello I am in display method
Hello I am inside a destructor

Process exited after 0.1613 seconds with return value 0
Press any key to continue . . .

8.WORKING WITH OPERATOR OVERLOADING

INTRODUCTION

- **Operator overloading** refers to **overloading** of **one operator for many different purpose**.
- The purpose of operator overloading is to **provide a special meaning** of an **operator for a user-defined data type** (such as class, structure, etc.)
- C++ **tries to make** the **user-defined data types behave in much the same way as the built-in types**
- C++ permits us to add two variables (objects) of user-defined types with the same syntax that is applied to the basic types
- For example, Addition operator can work on operands of type char, int, float and double. However, **if s1, s2, s3 are objects of the class**, then we can write the statement, s3=s1+s2;
- This means C++ has the ability to provide the operators with a special meaning for a data type

- Mechanism of giving **special meaning** to an **operator** is known as **operator overloading**

➢ **Operator -** is a **symbol** that indicates an **operation**.

➢ **Overloading - assigning** different **meanings** to an operator, depending upon the context.

➢ When an **operator is overloaded**, the produced symbol is called **operator function name**.

- The general form of an operator function is

```
return-type  operator  symbol(argument-list)
   {
       function body
    }
```

OR

```
return-type operator  symbol(argument-list);   //declaration

return-type class-name : : operator symbol (argument-list)
{
   function body                          //task defined
 }
```

Where,

- **return-type is the type of value returned by the specific operation.**
- **symbol (+ , - ,++ ,-- , etc) is the operator being overloaded.**
- **operator symbol is the function name, where operator is a keyword.**

Rules for overloading operator

- **Only the operators** which are **part of the C++ language** can be **overloaded**. **No new operator** can be created **using operator overloading**.
- You **can change the meaning of the operator** i.e., a + operator can be overloaded to perform multiplication operation or > operator can be overloaded to perform addition operation. **But you cannot change the priority of the operators.**
- The overloaded operator must have **at least one operand** that is of user-defined data type

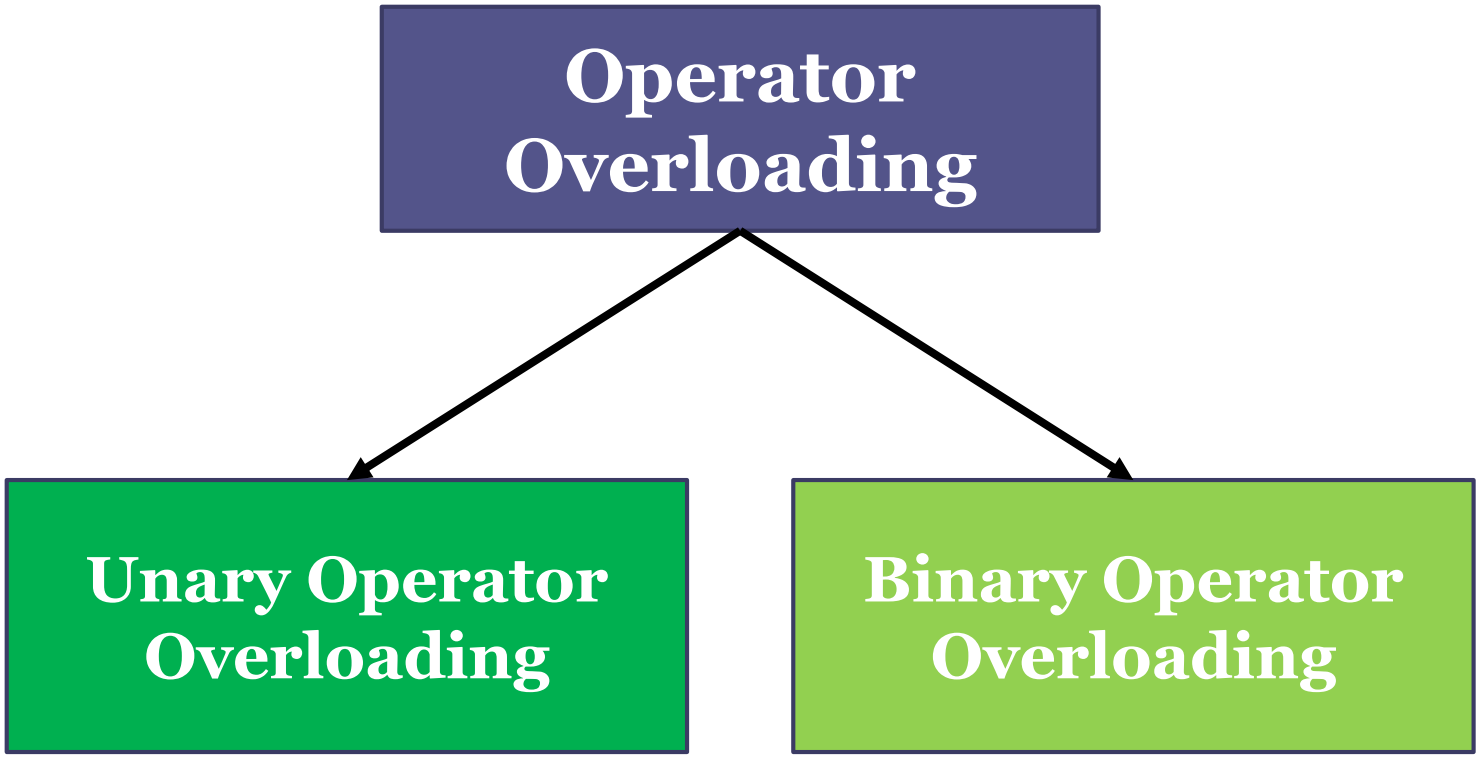
- Operator function should be either **member function** or **friend function**.
- **Friend function** requires **one argument** for **unary operator** and **two** for **binary operators**.
- **Member function** requires **zero arguments** for **unary operator** and **one** for **binary operators**.

- Operators which cannot be overloaded are
  - The Class member access operator
    - (.) [dot] is called as membership operator
    - (.*) [dot- asterisk] is called as pointer-to-member operator
  - The scope resolution operator (: :)
  - The sizeof() operator
  - The conditional operator (?:)

- Process of overloading involves the following steps:

1. **Creates the class** that defines the data type .i.e. to be used in the overloading operation.
2. **Declare the operator function** operator symbol() in the **public part of the class**. It may be either a member function or friend function.
3. **Define the operator function** to implement the required operations.

ways of operator overloading



Overloading unary operator

- The operator ++ (increment operator), - - (decrement operator) and – (minus) are unary operators
- Unary operators have only **one operand**
- ++ (increment operator) and - - (decrement operator) can be used as **prefix** or **suffix** with the function
- **Member function** requires **zero argument** for **unary operator**
- Increment ++ and decrement -- operator are overloaded in best possible way, i.e., increase the value of a data member by 1 if ++ operator operates on an object and decrease value of data member by 1 if -- operator is used

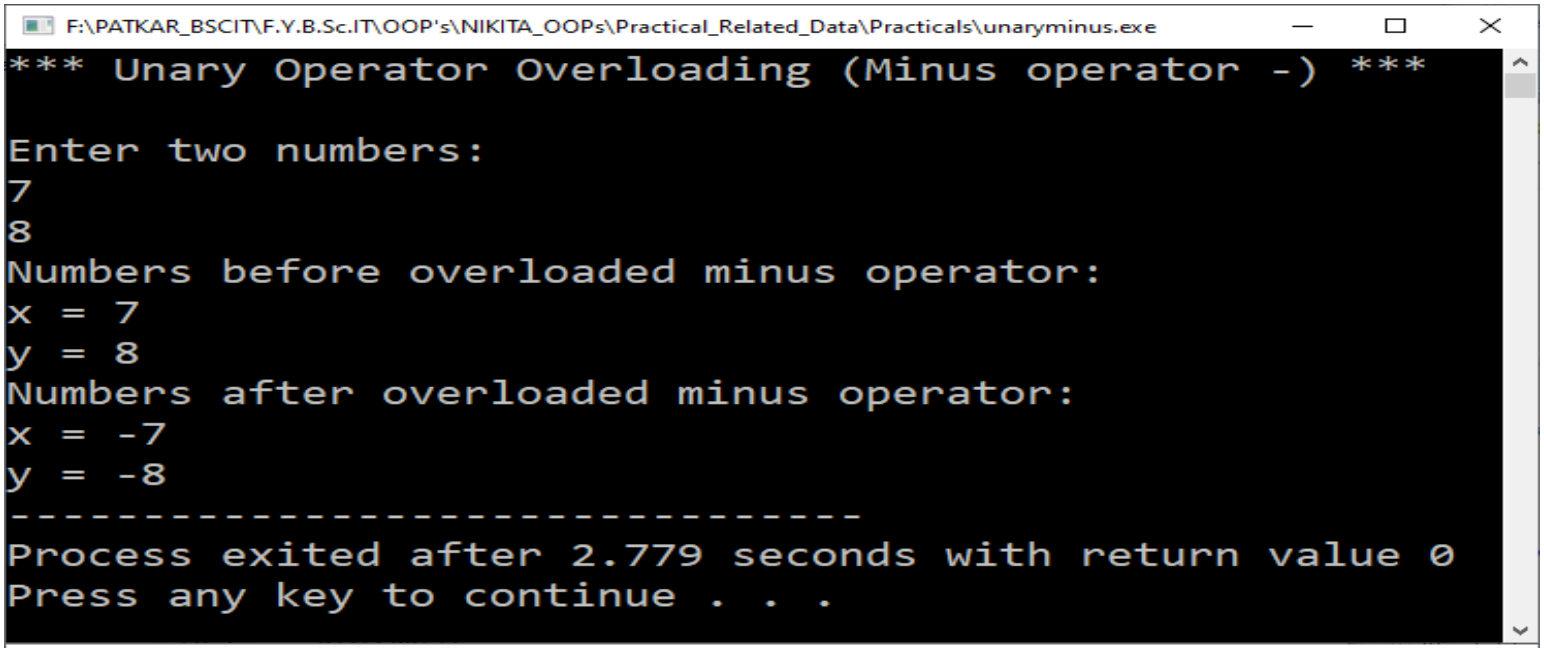
➢ Overloading unary operator : minus (-)

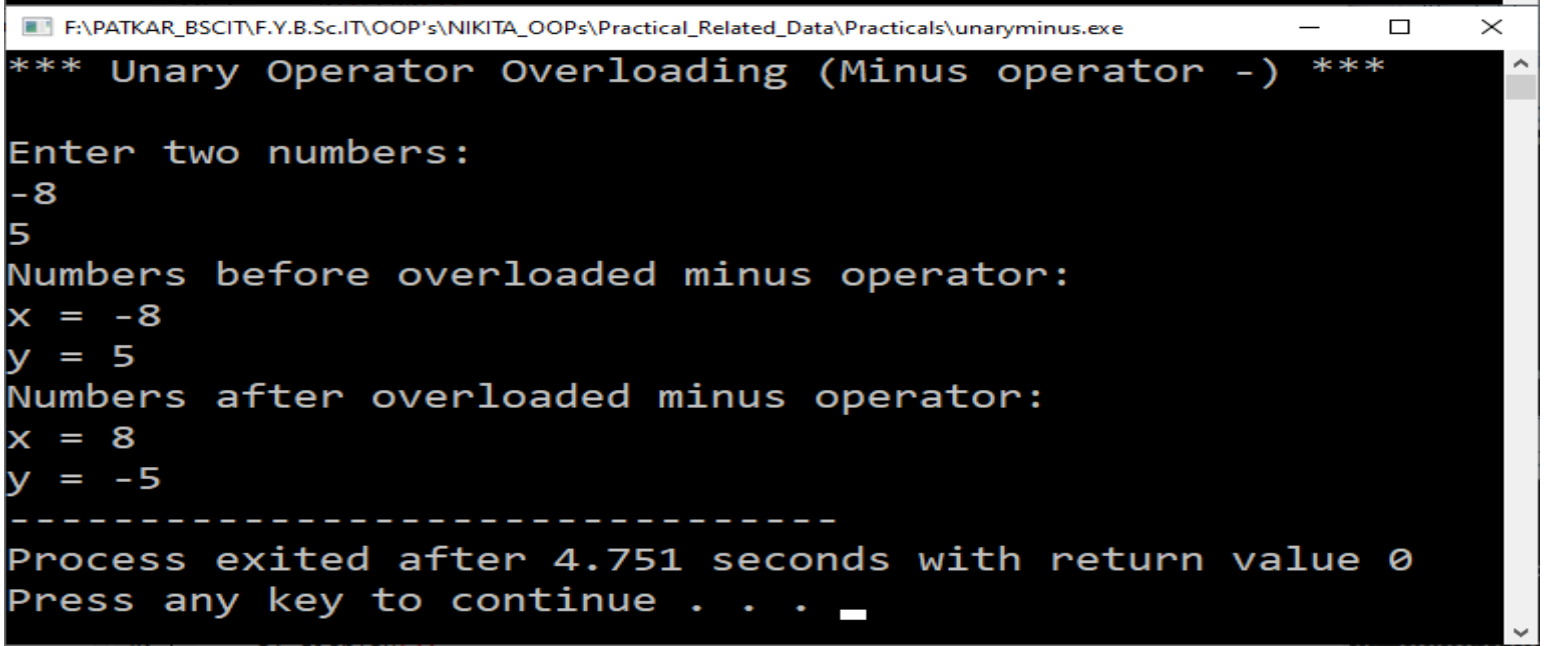
E.g.

unaryminus.cpp

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

class abc
{
     int x,y;
     public:
          void read()
          {
               cout<<"\n\nEnter two numbers:";
               cin>>x>>y;

          }
          void operator -()      //overloaded unary (minus) operator
          {
               x = -x;
               y = -y;
          }
          void display()
          {
               cout<<"\nx = "<< x <<endl<<"y = "<<y;
          }
};
int main()
{
     cout<<"*** Unary Operator Overloading (Minus operator -) ***";
     abc a1;
     a1.read();
     cout<<"Numbers before overloaded minus operator:";
     a1.display();

     -a1;                    // call unary minus operator function

     cout<<"\nNumbers after overloaded minus operator:";
     a1.display();
     return 0;
}
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\unaryminus.exe
*** Unary Operator Overloading (Minus operator -) ***
Enter two numbers:
7
8
Numbers before overloaded minus operator:
x = 7
y = 8
Numbers after overloaded minus operator:
x = -7
y = -8
Process exited after 2.779 seconds with return value 0
Press any key to continue . . .

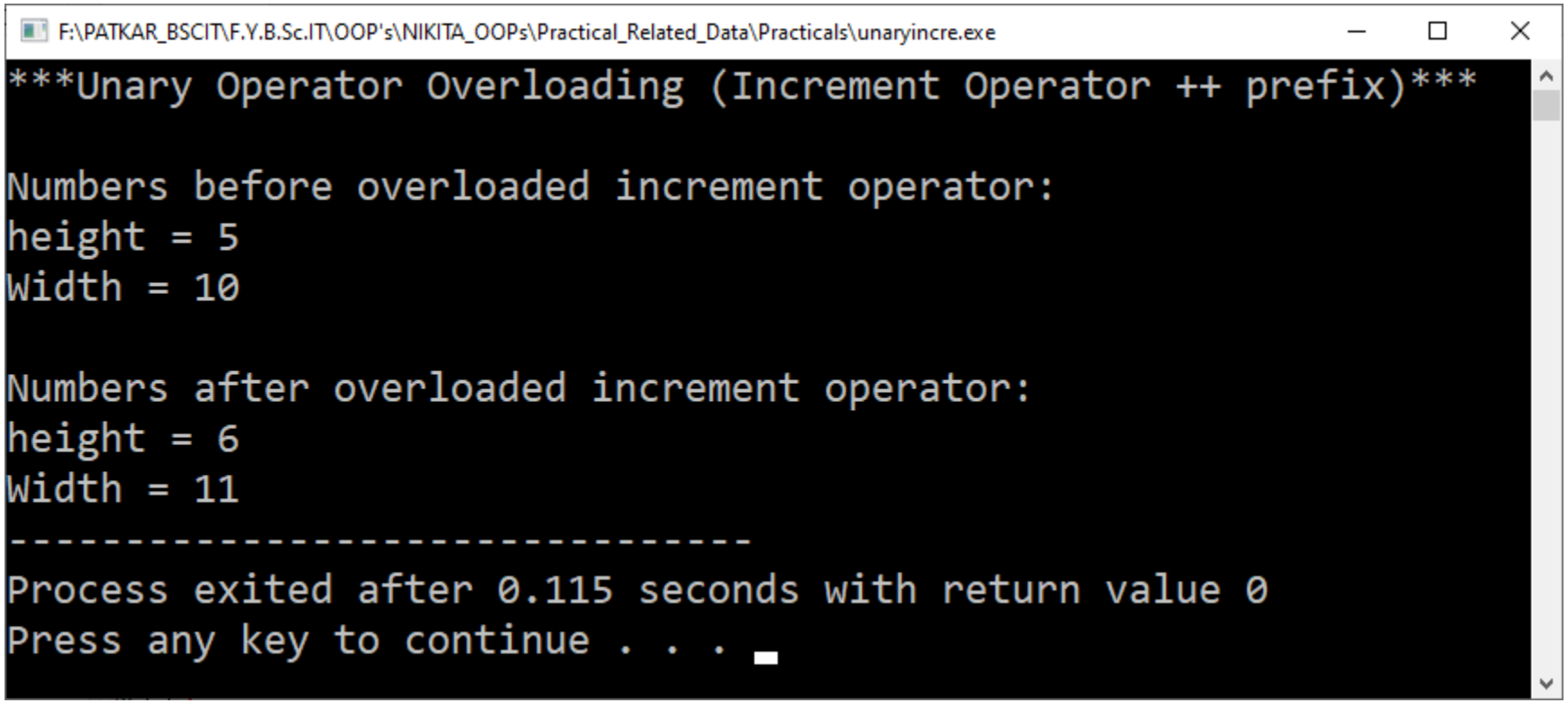

F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\unaryminus.exe
*** Unary Operator Overloading (Minus operator -) ***
Enter two numbers:
-8
5
Numbers before overloaded minus operator:
x = -8
y = 5
Numbers after overloaded minus operator:
x = 8
y = -5
Process exited after 4.751 seconds with return value 0
Press any key to continue . . .

➢ Overloading unary operator : increment prefix(++)

unaryincre.cpp

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

class test
{
    int h,w;
    public:
        test()
        {
            h=5;
            w=10;
        }
        void show()
        {
            cout<<"\nheight = "<<h<<"\nWidth = "<<w;
        }
        void operator ++ ( )  //overloaded unary (increment) operator
        {
            ++h;
            ++w;
        }

};
int main()
{
    cout<<"***Unary Operator Overloading (Increment Operator ++ prefix)***";
    test a1;
    cout<<"\n\nNumbers before overloaded increment operator:";
    a1.show();

    ++a1;                   // call unary (increment) operator function

    cout<<"\n\nNumbers after overloaded increment operator:";
    a1.show();
    return 0;
}
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\unaryincre.exe
Unary Operator Overloading (Increment Operator ++ prefix)
Numbers before overloaded increment operator:
height = 5
Width = 10
Numbers after overloaded increment operator:
height = 6
Width = 11

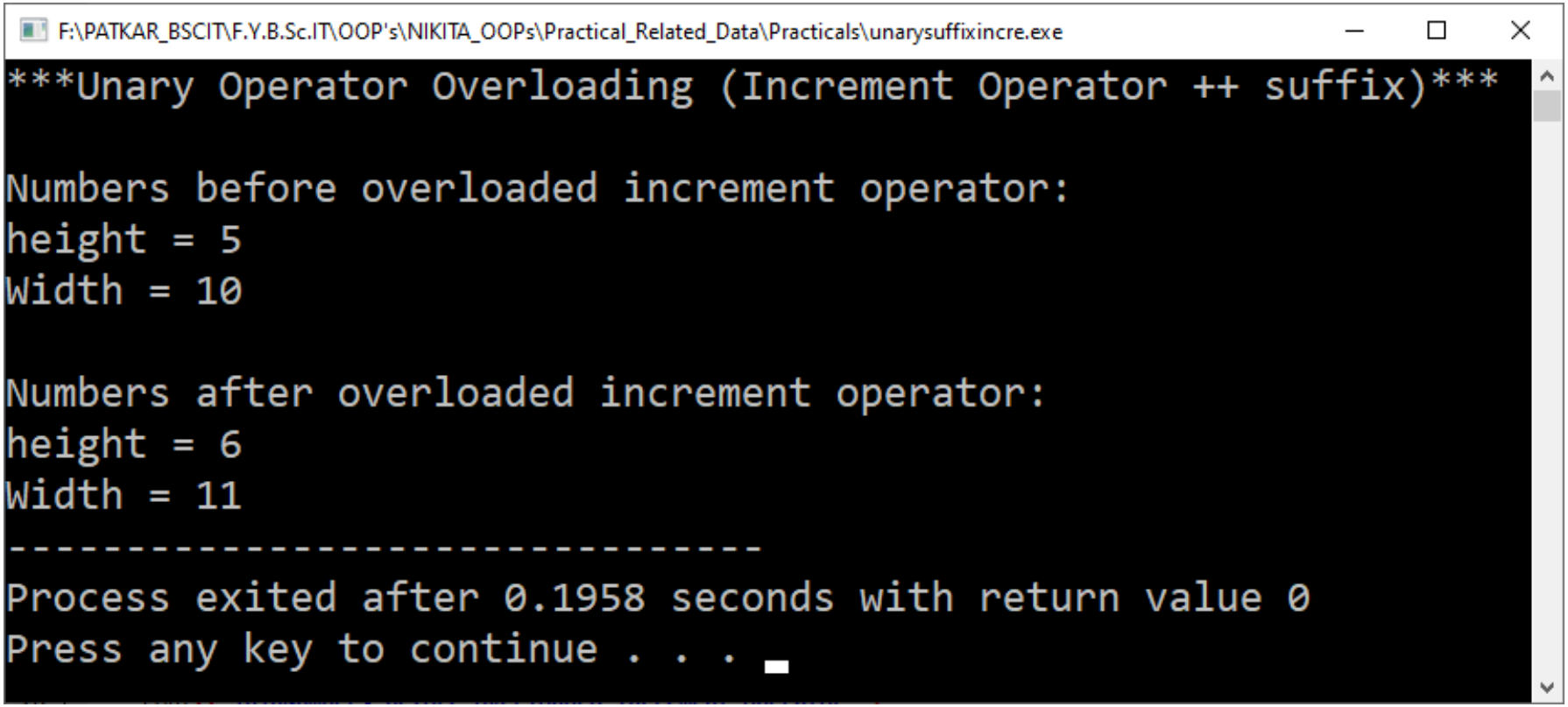
Process exited after 0.115 seconds with return value 0
Press any key to continue . . .

➢ Overloading unary operator : increment suffix (++)

unarysuffixincre.cpp

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

class test
{
    int h,w;
    public:
        test()
        {
            h=5;
            w=10;
        }
        void show()
        {
            cout<<"\nheight = "<<h<<"\nWidth = "<<w;
        }
        void operator ++ (int )  //overloaded unary (increment) operator
        {
            h++;
            w++;
        }

};
int main()
{
    cout<<"***Unary Operator Overloading (Increment Operator ++ suffix)***";
    test a1;
    cout<<"\n\nNumbers before overloaded increment operator:";
    a1.show();

    a1++;                   // call unary (increment) operator function

    cout<<"\n\nNumbers after overloaded increment operator:";
    a1.show();
    return 0;
}
```

- Output:



NOTE: The operator symbol for both **prefix(++i)** and **suffix(i++)** are the **same**. Hence, **we need two different function definitions to distinguish between them**. This is **achieved by passing a dummy int parameter** in the **suffix version**.

Overloading binary operator

- Those **operators** which operate on **two operands** or data are called **binary operators**
- **Member function** requires **one argument** and **Friend function** requires **two argument** for **binary operators**.
- There are two different categories of binary operators

➢ Arithmetic operators

➢ Assignment operators

overloading arithmetic operator

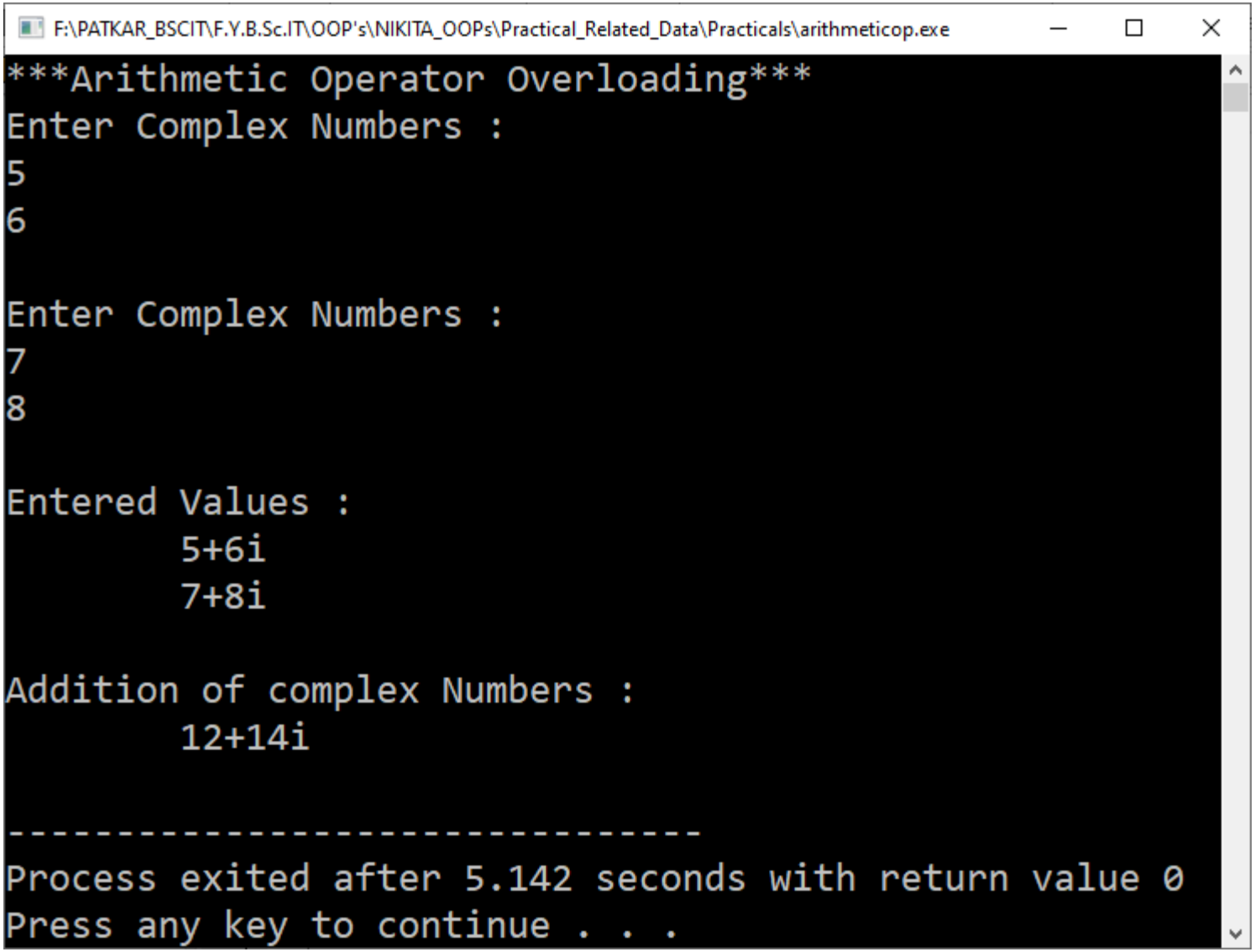
- Arithmetic operator (+,-,*,/,etc.) are most commonly used operator in C++.
- Almost all arithmetic operator can be overloaded to perform arithmetic operation on user-defined data type.

E.g.

arithmeticop.cpp

```cpp
#include<iostream>
using namespace std;
class Complex
{
    int num1, num2;
    public:
        void accept()
        {
            cout<<"\nEnter Complex Numbers : ";
            cin>>num1>>num2;
        }
        Complex operator + (Complex obj)   //Overloading '+' operator
        {
            Complex c;
            c.num1=num1+obj.num1;
            c.num2=num2+obj.num2;
            return c;
        }
        void display()
        {
            cout<<num1<<"+"<<num2<<"i"<<"\n";
        }
};
int main()
{
        cout<<"***Arithmetic Operator Overloading***";
        Complex c1, c2, sum;      //Created Object of Class Complex i.e c1 and c2
        c1.accept();             //Accepting the values
        c2.accept();

        sum = c1+c2;   //Addition of object (will call arithmetic (+) opertor function)

        cout<<"\nEntered Values : \n";
        cout<<"\t";
        c1.display();    //Displaying user input values
        cout<<"\t";
        c2.display();

        cout<<"\nAddition of complex Numbers : \n";
        cout<<"\t";
        sum.display();  //Displaying the addition of real and imaginary numbers
        return 0;
}
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\arithmeticop.exe
Arithmetic Operator Overloading
Enter Complex Numbers :
5
6
Enter Complex Numbers :
7
8
Entered Values :
5+6i
7+8i
Addition of complex Numbers :
12+14i

Process exited after 5.142 seconds with return value 0
Press any key to continue . . .

overloading assignment operator

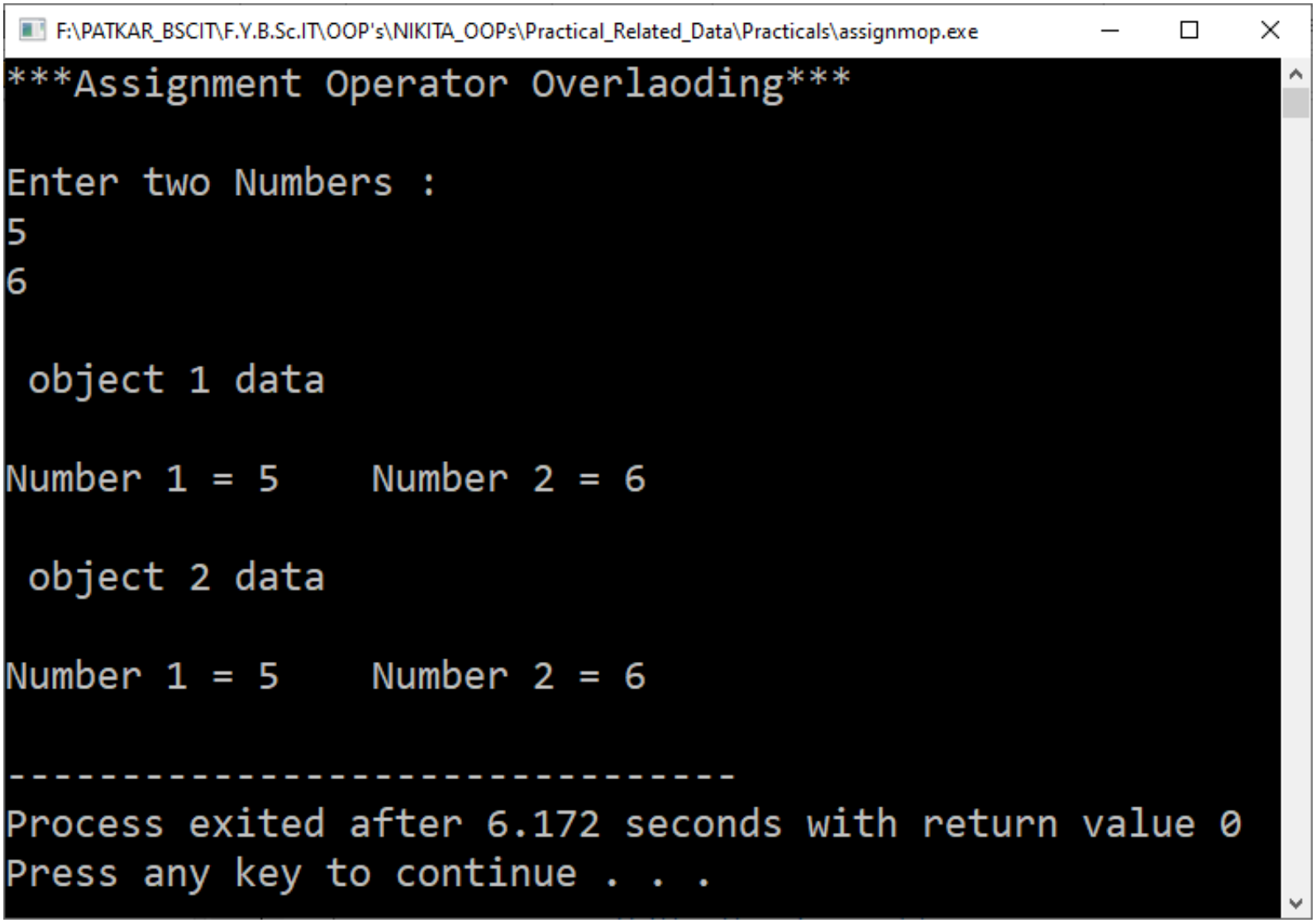
- By overloading assignment operator, **all values of one object** (i.e. , instance variables) **can be copied to another object**.

E.g.

assignmop.cpp

```
1   #include<iostream>
2   using namespace std;
3   class abc
4   {
5       int num1, num2;
6       public:
7           void accept()
8           {
9               cout<<"\n\nEnter two Numbers : ";
10              cin>>num1>>num2;
11          }
12          void display()
13          {
14              cout<<"\nNumber 1 = "<<num1<<"\tNumber 2 = "<<num2<<"\n";
15          }
16          void operator = (abc a)    //Overloading '=' operator
17          {
18              num1=a.num1;
19              num2=a.num2;
20          }
21  };
22  int main()
23  {
24          cout<<"***Assignment Operator Overlaoding***";
25          abc c1, c2;
26          c1.accept();
27          cout<<"\n object 1 data \n";
28          c1.display();
29
30          c2=c1;              //will call assignment (=) operator
31
32          cout<<"\n object 2 data \n";
33          c2.display();
34          return 0;
35  }
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\assignmop.exe
Assignment Operator Overlaoding
Enter two Numbers :
5
6
object 1 data
Number 1 = 5 Number 2 = 6
object 2 data
Number 1 = 5 Number 2 = 6
Process exited after 6.172 seconds with return value 0
Press any key to continue . . .

Overloading using friend function

- In addition to member function, an operator function can be defined as a friend function of the class for which it is being overloaded.
- The **operator function defined** as a **friend function** of the class is **known as the friend operator function**.
- **Like other friend function** of the class, the **friend operator function** is *declared inside the class*, however defined outside the class.

overloading unary operator using friend function

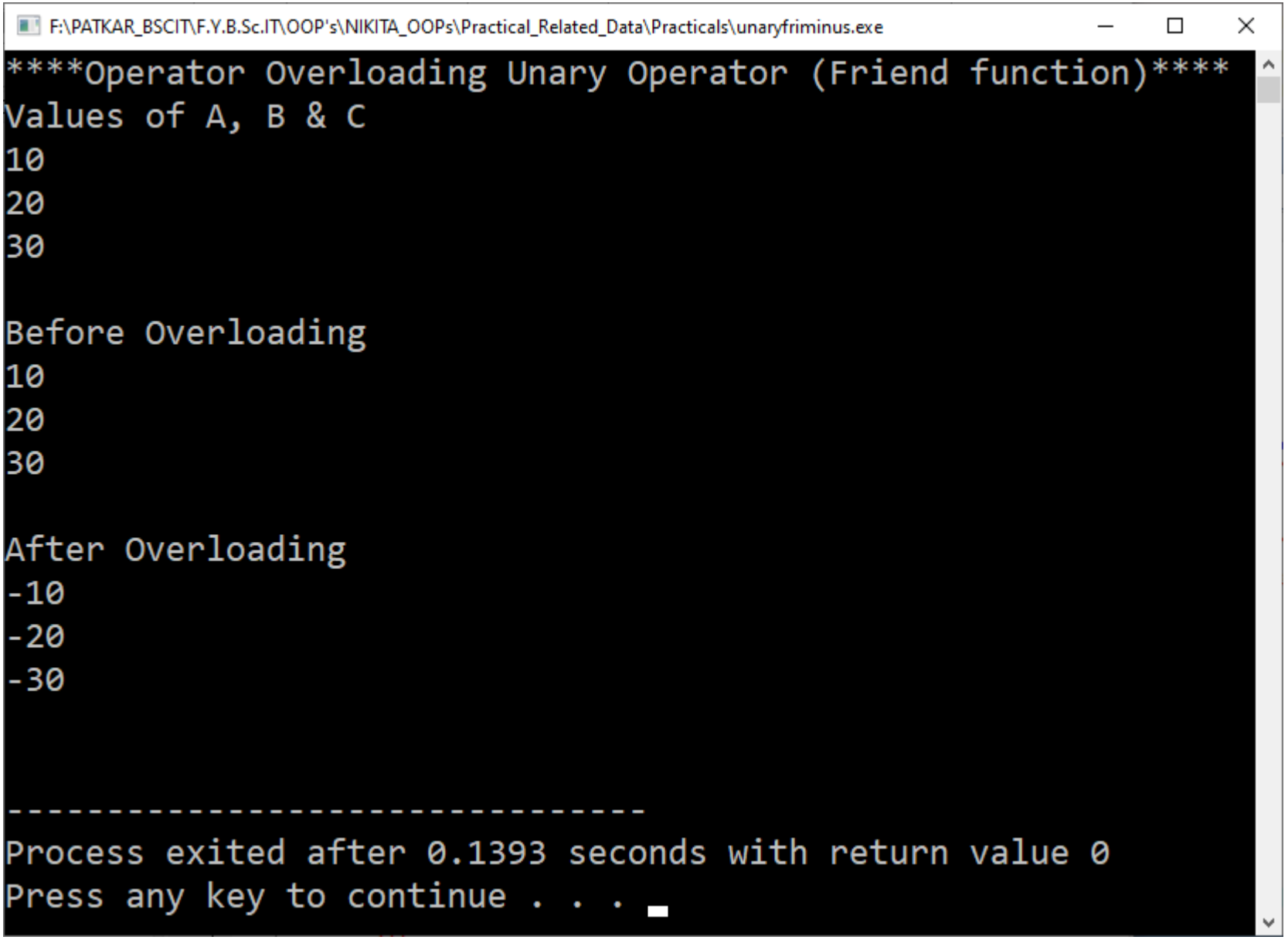
- **Friend function** requires **one argument** for **unary operator**

E.g.

unaryfriminus.cpp

```cpp
#include<iostream>
using namespace std;
class abc
{
    int a=10;
    int b=20;
    int c=30;
    public:
        void getvalues()
        {
            cout<<"\nValues of A, B & C\n";
            cout<<a<<"\n"<<b<<"\n"<<c<<"\n"<<endl;
        }
        void show()
        {
            cout<<a<<"\n"<<b<<"\n"<<c<<"\n"<<endl;
        }
        void friend operator - (abc &x);
};
void operator - (abc &x)
{
    x.a = -x.a;
    x.b = -x.b;
    x.c = -x.c;
}
int main()
{
    cout<<"****Operator Overloading Unary Operator (Friend function)****";
    abc x1;
    x1.getvalues();
    cout<<"Before Overloading\n";
    x1.show();
    cout<<"After Overloading \n";
    -x1;
     x1.show();
     return 0;
}
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\unaryfriminus.exe
****Operator Overloading Unary Operator (Friend function)****
Values of A, B & C
10
20
30
Before Overloading
10
20
30
After Overloading
-10
-20
-30

Process exited after 0.1393 seconds with return value 0
Press any key to continue . . .

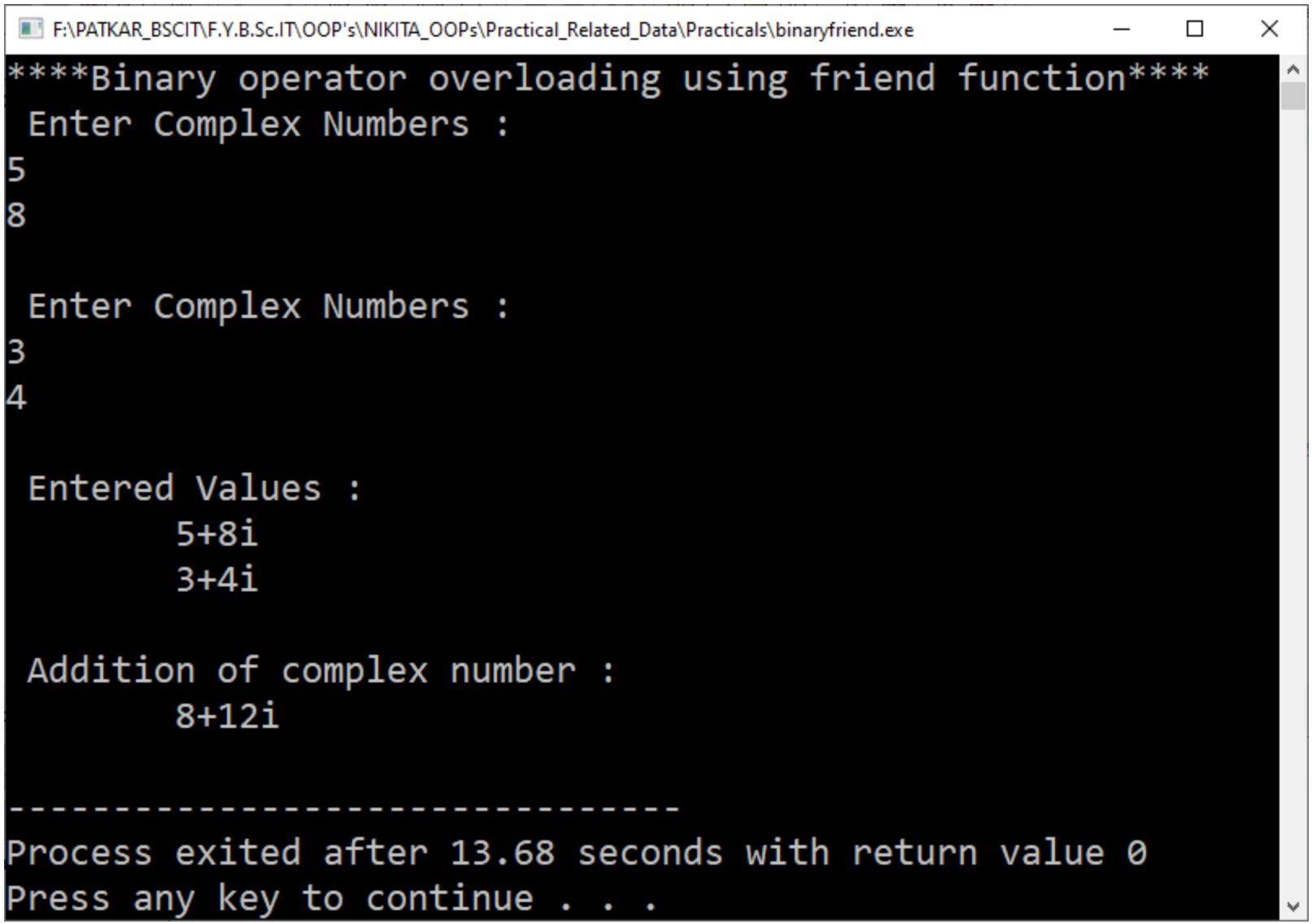
overloading binary operator using friend function

- **Friend function** requires **two argument** for **binary operators**

binaryfriend.cpp

```cpp
#include<iostream>
using namespace std;
class complex
{
    int num1, num2;
    public:
        void accept()
        {
            cout<<"\n Enter Complex Numbers : ";
            cin>>num1>>num2;
        }
        void display()
        {
            cout<<num1<<"+"<<num2<<"i"<<"\n";
        }
        friend complex operator + (complex c1, complex c2);     //Overloading '+' operator using Friend function
};
complex operator + (complex c1, complex c2)
{
        complex c;
        c.num1=c1.num1+c2.num1;
        c.num2=c1.num2+c2.num2;
        return c;
}
int main()
{
        cout<<"****Binary operator overloading using friend function****";
        complex c1,c2, sum;
        c1.accept();
        c2.accept();

        sum = c1+c2;

        cout<<"\n Entered Values : \n";
        cout<<"\t";
        c1.display();    //Displaying user input values
        cout<<"\t";
        c2.display();

        cout<<"\n Addition of complex number : \n";
        cout<<"\t";
        sum.display();  //Displaying the addition of real and imaginary numbers
        return 0;
}
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\binaryfriend.exe
****Binary operator overloading using friend function****
Enter Complex Numbers :
5
8
Enter Complex Numbers :
3
4
Entered Values :
5+8i
3+4i
Addition of complex number :
8+12i

Process exited after 13.68 seconds with return value 0
Press any key to continue . . .

type conversion

- C++ allows you to covert **one data type** to **another data type** and hence it is called **type conversion**
- E.g.: float → int
- For example

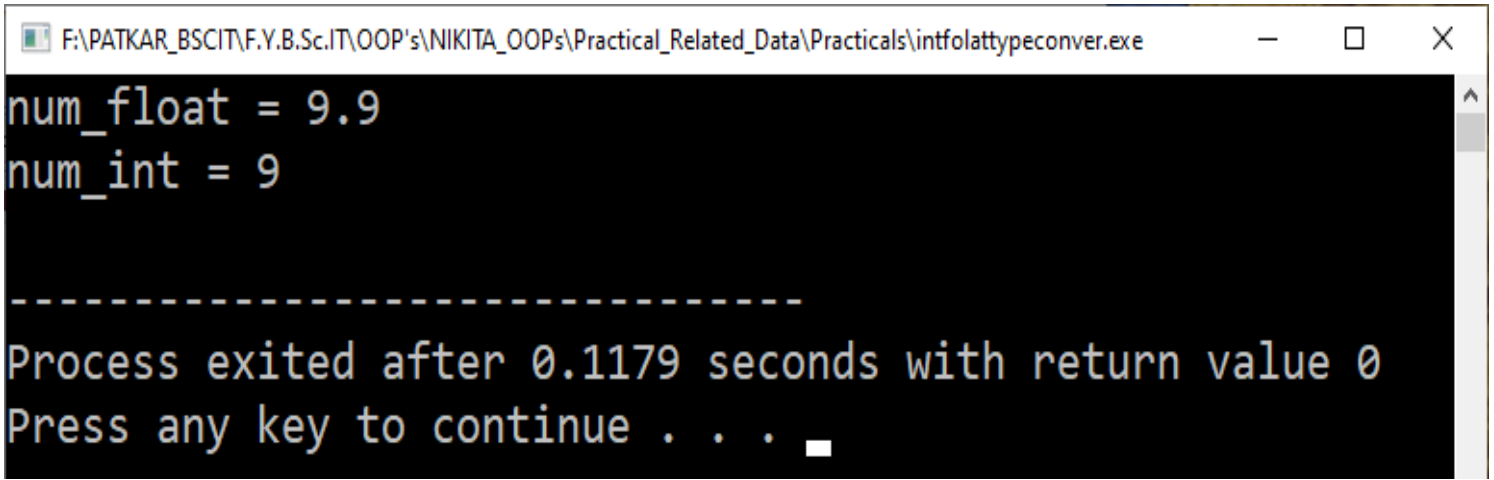
```
int m;
float x=3.1419;
m=x;
```

- Convert x to an integer before its values is assigned to m. Thus, fractional part is truncated
- C++ already **knows** how to **convert between built-in data types** i.e. C++ provides mechanism to perform automatic type conversion if all variable are of basic type.

E.g. Conversion between built-in data types.

intfolattypeconver.cpp

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

int main()
{

    int num_int;
    float num_float=9.9;

    // assigning a float value to an int variable
    num_int=num_float;

    cout << "num_float = " << num_float << endl;
    cout << "num_int = " << num_int << endl;

    return 0;
}
```



- However, it **does not know** how to convert **any user-defined data types** . i.e. in situations where one of the operand is an object and the other is built-in variable or if they belong to two different classes.
- There are **three** possibilities of user defined data type conversion. They are as follows:

1. Conversion from **basic-data** type to **user-defined (class type)** data type.
2. Conversion from **class** type to **basic-data** type.
3. Conversion from **one class** type to **another class** type.

conversion from basic to class data type

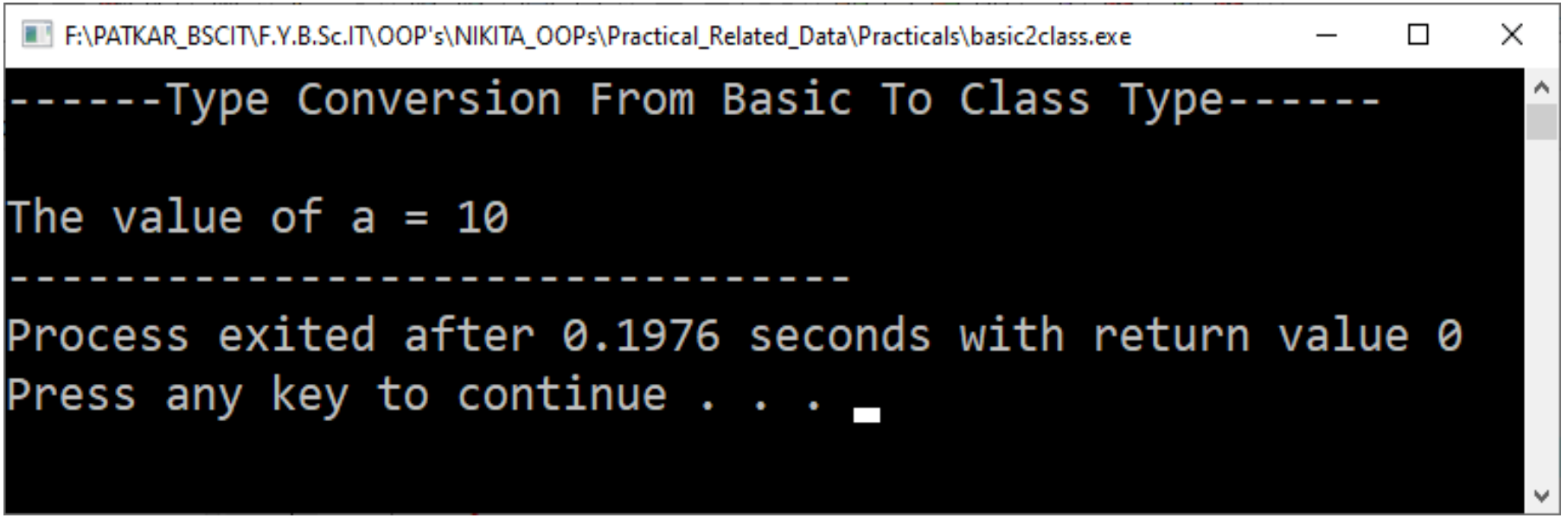
- In this conversion, the **from** type is **primitive data type (basic data type)** and the **to** type is **class object**
- Conversion from basic to class type is easily carried out.
- To perform this conversion, the idea is to use the **constructor** to perform type conversion during the object creation. (**This conversion is achieved using constructor**)
- **Left-operand** of **(=)** sign is always **class type** and **right-hand operand** is always **basic type**.

E.g.

basic2class.cpp

```
1  #include<iostream>
2  using namespace std;
3  class sample
4  {
5      int a;
6      public:
7          sample()
8          {
9              a=0;
10         }
11         sample(int x)
12         {
13             a=x;
14         }
15         void show()
16         {
17             cout<<"\n\nThe value of a = "<<a;
18         }
19 };
20 int main()
21 {
22     cout<<"------Type Conversion From Basic To Class Type------";
23     sample s;       //object is created
24     int m=10;
25
26     s = m;          // conversion of int type to class type.
27                      //here s (object) is class type  and m (int value) is basic type
28     s.show();
29     return 0;
30 }
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\basic2class.exe
------Type Conversion From Basic To Class Type------
The value of a = 10

Process exited after 0.1976 seconds with return value 0
Press any key to continue . . .

conversion from class to basic data type

- In this conversion, the programmer explicitly tells the compiler to perform the conversion to the basic type
- In this conversion, the **from** type is a **class object** and the **to** type is **primitive data type (basic data type)**
- These instructions are written in a member function.
- Such function is known as **overloading of type cast operators**
- C++ allows to define an **overloaded casting operator** that could be used to convert a **class type data to a basic type (This conversion is done using casting operator function)**
- The normal form of an **overloaded casting operator function**, also known as **a conversion function**

- **Left-operand** of **(=)** sign is always **basic type** and **right-hand operand** is always **class type**
- The function converts the class type data to **typename (basic data type) i.e. [e.g. the operator int() converts a class type object to type int]**

➢ The casting operator function should satisfy the following conditions:

1. It **should** be **class member function**
2. It must **not** mention a **return type**
3. The conversion function should **not have any argument**

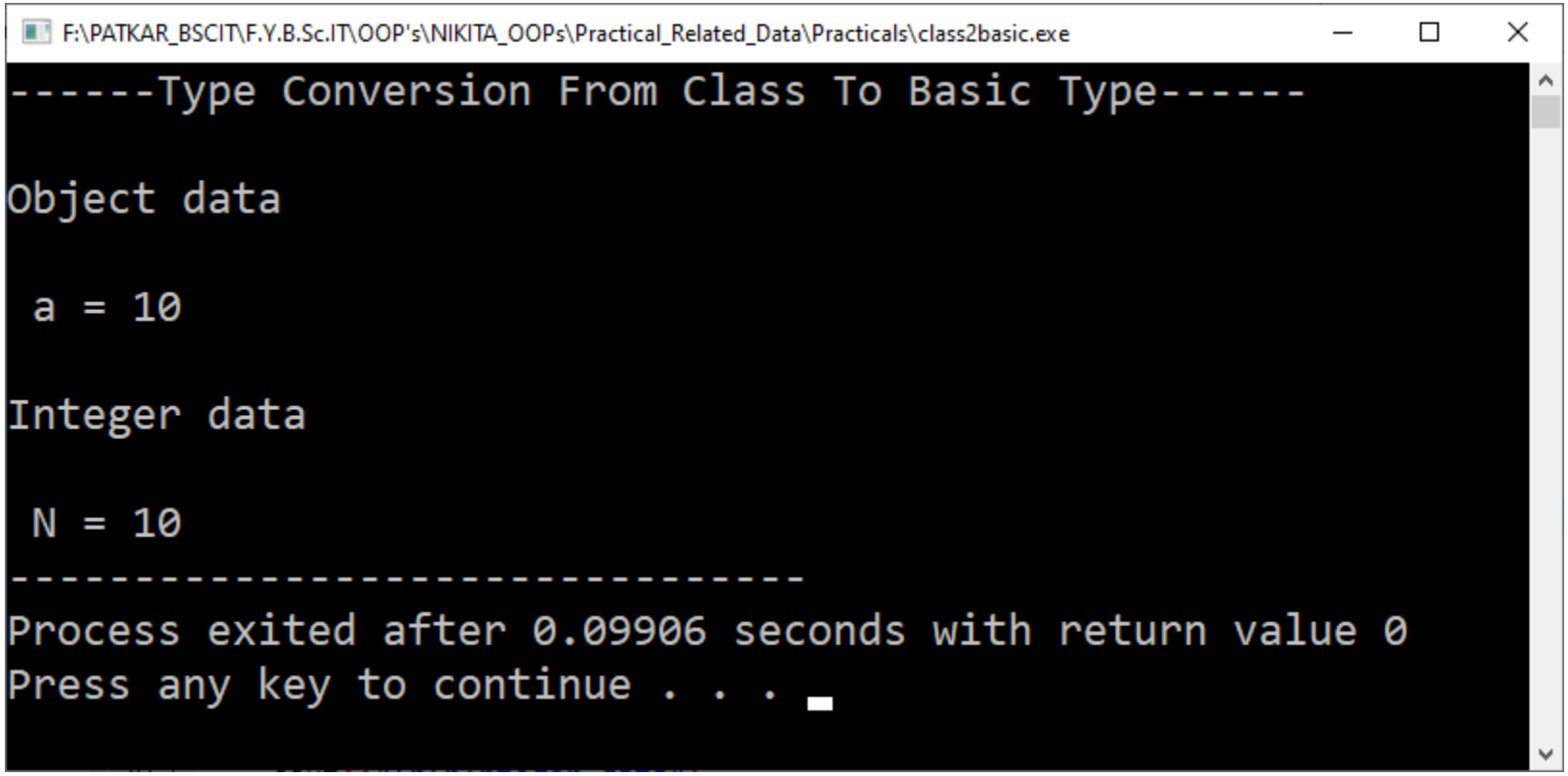
- Syntax:

```
operator  typename( )
{
        function statement;
}
```

class2basic.cpp

```cpp
#include<iostream>
using namespace std;
class abc
{
    int a;
    public:
        abc(int x)
        {
            a=x;
        }
        void show()
        {
            cout<<"\n\n a = "<<a;
        }
        operator int()          //casting operator function
        {
            return a;
        }
};
int main()
{
    cout<<"------Type Conversion From Class To Basic Type------";
    abc z(10);          //object is created
    int n;

    cout<<"\n\nObject data";
    z.show();

    cout<<"\n\nInteger data";
    n = z;                // conversion of class  type to basic  type.
                          //here z (object) is class type  and n (int value) is basic type
    cout<<"\n\n N = "<<n;
    return 0;
}
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\class2basic.exe
------Type Conversion From Class To Basic Type------
Object data
a = 10
Integer data
N = 10

Process exited after 0.09906 seconds with return value 0
Press any key to continue . . .

conversion from one class type to another class type data type

- In this one class type is to be converted into another class type
- The **class type (object)** that appears on the **right-hand side** is known as **source class** and the **class type (object)** that appears on the **left-hand side** is known as the **destination class**
- The conversion of **one class type to another** can be handled by **using** either the **constructor** or the **conversion function (casting operator)**
- The compiler treats both of them in the same way . However , if the **constructor** is used for conversion ,it must be **defined in** the **destination class** AND if the **casting operator function** is used , it must be **defined in** the **source class**
- e.g.

obj1=obj2; // objects of two different classes

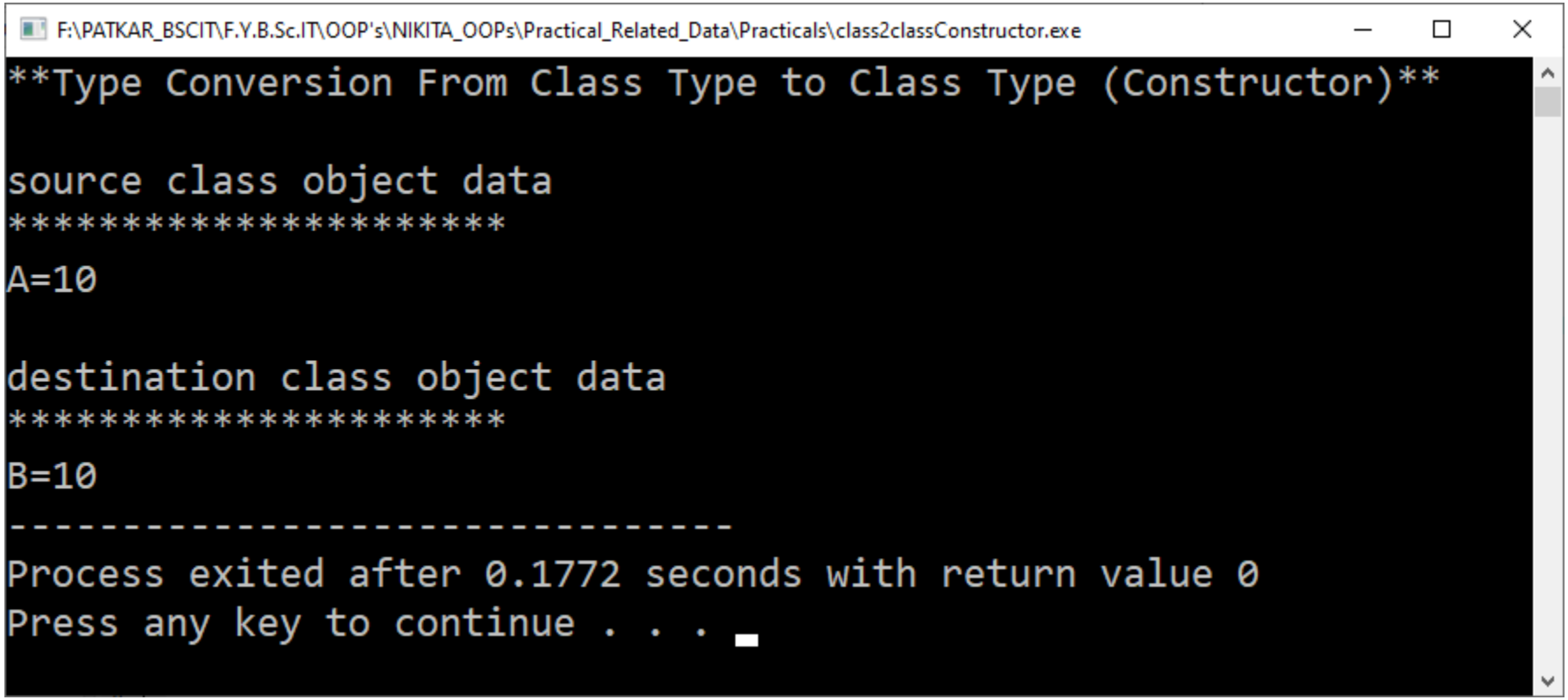
class type to class type (using constructor)

- When the **constructor** is used for conversion ,it must be **defined in** the **destination class**

class2classConstructor.cpp

```cpp
#include <iostream>
using namespace std;
class one_class         // Source class
{
    int a;
public:
    one_class(int x)
    {
        a=x;
    }
    void show()
    {
        cout<<"A="<<a;
    }
    int get()
    {
        return a;
    }
};
class two_class      // Destination class (constructor defined)
{
    int b;
public:
    two_class(one_class ab)
    {
        b=ab.get();
    }
    void show()
    {
        cout<<"B="<<b;
    }
};
int main()
{
    cout<<"**Type Conversion From Class Type to Class Type (Constructor)**";
    one_class a1(10);                // Creating object of class one_class
    cout<<"\n\nsource class object data";
    cout<<"\n**********************\n";
    a1.show();                      // Displaying data of object of class one_class
    cout<<"\n\ndestination class object data";
    cout<<"\n**********************\n";
    two_class b1 =a1;                                   // Creating object of class two_class // Class type conversion
    b1.show();                        // Displaying data of object of class two_class
    return 0;          }
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\class2classConstructor.exe
Type Conversion From Class Type to Class Type (Constructor)
source class object data
****
A=10
destination class object data
****
B=10
Process exited after 0.1772 seconds with return value 0
Press any key to continue . . .

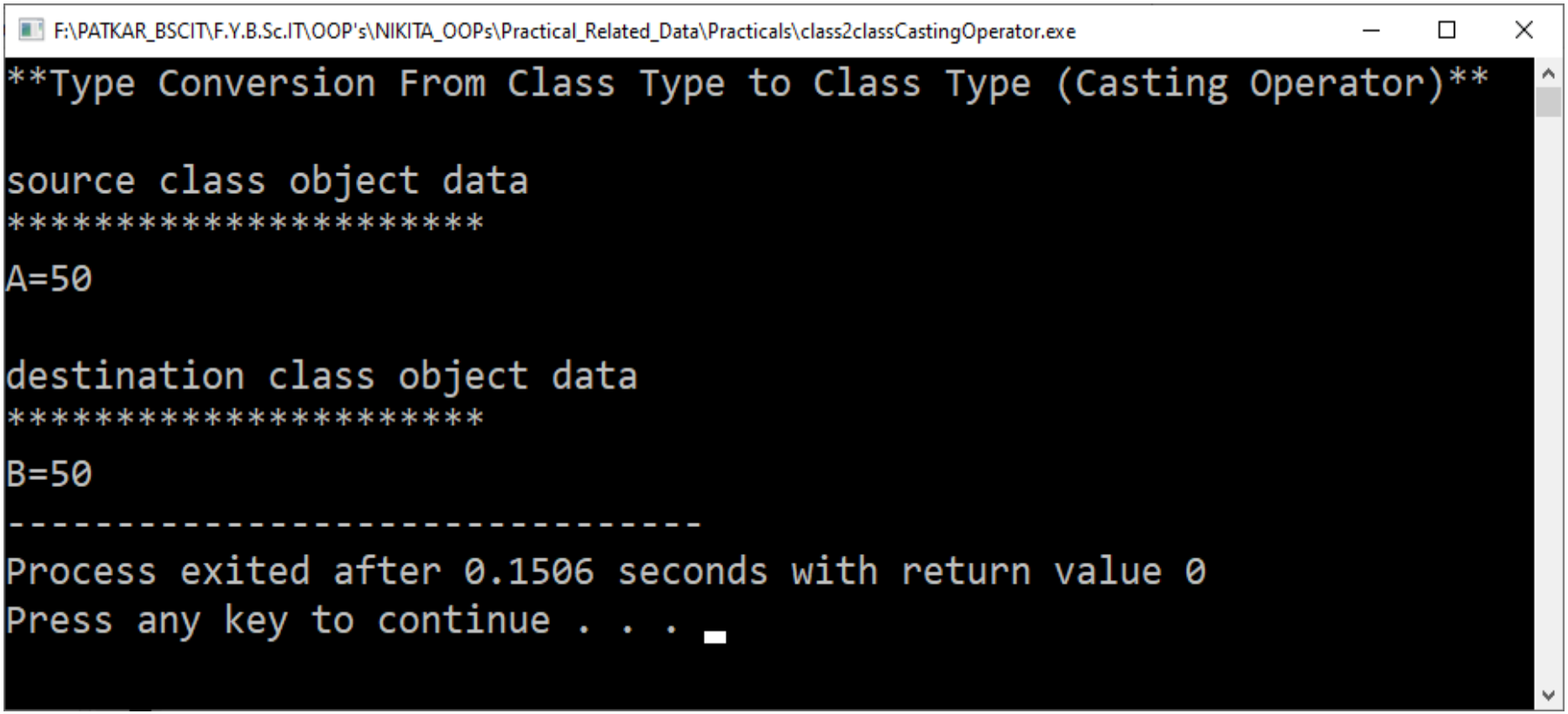
class type to class type (using casting operator function)

- When the **casting operator function** is used , it must be **defined in** the **source class**
- In the case of casting operator function
- **operator** typename()
- {
- .....
- }
- In the case of conversion between objects, **typename** refers to the **destination class**

class2classCastingOperator.cpp

```cpp
#include <iostream>
using namespace std;
class two_class      // Destination class
{
    int b;
public:
    two_class(int y)
    {
        b=y;
    }
    void show()
    {
        cout<<"B="<<b;
    }
};
class one_class        // Source class
{
    int a;
public:
    one_class(int x)
    {
        a=x;
    }
    void show()
    {
        cout<<"A="<<a;
    }
    operator two_class()              // casting opearator function
    {
        return two_class(a);
    }
};
int main()
{
    cout<<"**Type Conversion From Class Type to Class Type (Casting Operator)**";
    one_class a1(50);                // Creating object of class one_class
    cout<<"\n\nsource class object data";
    cout<<"\n**********************\n";
    a1.show();                      // Displaying data of object of class one_class
    cout<<"\n\ndestination class object data";
    cout<<"\n**********************\n";
    two_class b1 =a1;                                  // Creating object of class two_class  // Class type to class type  conversion
    b1.show();                       // Displaying data of object of class two_class
    return 0;       }
```

- Output:


F:\PATKAR_BSCIT\F.Y.B.Sc.IT\OOP's\NIKITA_OOPs\Practical_Related_Data\Practicals\class2classCastingOperator.exe
Type Conversion From Class Type to Class Type (Casting Operator)
source class object data
****
A=50
destination class object data
****
B=50
Process exited after 0.1506 seconds with return value 0
Press any key to continue . . .