

UNIT 1

Basic Concepts of OOPS

Before starting to learn C++ it is essential that one must have a basic knowledge of the concepts of Object Oriented Programming.

Some of the important object oriented features are namely:

- Objects
- Classes
- Inheritance
- Data Abstraction
- Data Encapsulation
- Polymorphism
- Overloading
- Reusability

In order to understand the basic concepts in C++, the programmer must have a command of the basic terminology in object-oriented programming. Below is a brief outline of the concepts of Object-oriented programming languages:

Objects

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data. An Object is a collection of data members and associated member functions also known as methods.

Classes

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects. Characteristics of an object are represented in a class as **Properties (Attributes)**. The actions that can be performed by objects become functions of the class and are referred to as **Methods (Functions)**.

For example consider we have a Class of Cars under which Santro Xing, Alto and WaganR represents individual Objects. In this context each Car Object will have its own, Model, Year of Manufacture, Colour, Top Speed, Engine Power etc., which form **Properties** of the Car class and the associated actions i.e., object functions like Start, Move, Stop form the **Methods** of Car Class.

No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

Inheritance

Inheritance is the process of forming a new class from an existing class or base class. The base class is also known as parent class or super class. The new class that is formed is called derived class. Derived class is also known as a child class or sub class. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

Data Abstraction

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

Data Encapsulation

Data Encapsulation combines data and functions into a single unit called Class. When using Data Encapsulation, data is not accessed directly; it is only accessible through the functions present inside the class. Data Encapsulation enables the important concept of data hiding possible.

Polymorphism

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.

Overloading

Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an exiting operator or function begins to operate on new data type, or class, it is understood to be overloaded.

Reusability

This term refers to the ability for multiple programmers to use the same written and debugged existing class of data. This is a time saving device and adds code efficiency to the language. Additionally, the programmer can incorporate new features to the existing class, further developing the application and allowing users to achieve increased performance.

Introduction to C++

Variable, Constants and Data types in C++

Variables

A variable is the storage location in memory that is stored by its value. A variable is identified or denoted by a variable name. The variable name is a sequence of one or more letters, digits or underscore, for example: character _

Rules for defining variable name:

- A variable name can have one or more letters or digits or underscore for example character _.
- White space, punctuation symbols or other characters are not permitted to denote variable name.
- A variable name must begin with a letter..
- Variable names cannot be keywords or any reserved words of the C++ programming language.

C++ is a case-sensitive language. Variable names written in capital letters differ from variable names with the same name but written in small letters. For example, the variable name EXFORSYS differs from the variable name exforsys.

Data Types

Below is a list of the most commonly used Data Types in C++ programming language Using variable names and data type, we shall now learn how to declare variables.

Declaring Variables:

In order for a variable to be used in C++ programming language, the variable must first be declared. The syntax for declaring variable names is **data type variable name;**

The date type can be int or float or any of the data types listed above. A variable name is given based on the rules for defining variable name (refer above rules).

Example:
int a;
This declares a variable name a of type int.
If there exists more than one variable of the same type, such variables can be represented by separating variable names using comma.

For instance
int x,y,z ;
This declares 3 variables x, y and z all of data type int.

The data type using integers (int, short int, long int) are further assigned a value of **signed** or **unsigned**. Signed integers signify positive and negative number value. Unsigned integers signify only positive numbers or zero.
For example it is declared as
unsigned short int a;
signed int z;
By default, unspecified integers signify a signed integer.

For example:
int a;
is declared a signed integer
It is possible to initialize values to variables:
data type variable name = value;
Example:
int a=0;
int b=5;

Constants
Constants have fixed value. Constants, like variables, contain data type. Integer constants are represented as decimal notation, octal notation, and hexadecimal notation. Decimal notation is represented with a number. Octal notation is represented with the number preceded by a zero character. A hexadecimal number is preceded with the characters 0x.

Example
80 represent decimal
0115 represent octal
0x167 represent hexadecimal
By default, the integer constant is represented with a number.
The unsigned integer constant is represented with an appended character **u**. The long integer constant is represented with character **l**.

Example:
78 represent int
85u present unsigned int
78l represent long
Floating point constants are numbers with decimal point and/or exponent.
Example
2.1567
4.02e24
These examples are valid floating point constants.
Floating point constants can be represented with f for floating and l for double precision floating

point numbers.
Character constants have single character presented between single quotes.
Example
_c'
_a'
are all character constants.
Strings are sequences of characters signifying string constants. These sequence of characters are represented between double quotes.

Example:
—Exforsys Training!
is an example of string constant.
Referencing variables
The & operator is used to reference an object. When using this operator on an object, you are provided with a pointer to that object. This new pointer can be used as a parameter or be assigned to a variable.

C++ Objects and Classes

An Overview about Objects and Classes

In object-oriented programming language C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an object. A class is an extended concept similar to that of structure in C programming language, this class describes the data properties alone. In C++ programming language, class describes both the properties (data) and behaviors (functions) of objects. Classes are not objects, but they are used to instantiate objects.

Features of Class:

Classes contain member data and member functions. As a unit, the collection of member data and member functions is an object. Therefore, this unit of objects makes up a class.

How to write a Class:

In Structure in C programming language, a structure is specified with a name. The C++ programming language extends this concept. A class is specified with a name after the keyword class.

The starting flower brace symbol, {is placed at the beginning of the code. Following the flower brace symbol, the body of the class is defined with the member functions data.
Then the class is closed with a flower brace symbol} and concluded with a colon;.

```
class exforsys
{
member data;
member functions;
……
};
```

There are different access specifiers for defining the data and functions present inside a class.

Access specifiers:
Access specifiers are used to identify access rights for the data and member functions of the class. There are three main types of access specifiers in C++ programming language:

private
public
protected

 A private member within a class denotes that only members of the same class have accessibility. The private member is inaccessible from outside the class.
 Public members are accessible from outside the class.
 A protected access specifier is a stage between private and public access. If member functions defined in a class are protected, they cannot be accessed from outside the class but can be accessed from the derived class.
When defining access specifiers, the programmer must use the keywords: private, public or protected when needed, followed by a semicolon and then define the data and member functions under it.

```
class exforsys
{
private:
int x,y;
public:
void sum()
{
……
……
}
};
```

In the code above, the member x and y are defined as private access specifiers. The member function sum is defined as a public access specifier.

General Syntax of a class:
General structure for defining a class is:

```
class classname
{
acess specifier:
data member;
member functions;
acess specifier:
data member;
member functions;
};
```

Generally, in class, all members (data) would be declared as private and the member functions would be declared as public. Private is the default access level for specifiers. If no access specifiers are identified for members of a class, the members are defaulted to private access.

```
class exforsys
{
int x,y;
public:
void sum()
{
……
……
}
};
```

In this example, for members x and y of the class exforsys there are no access specifiers identified. exforsys would have the default access specifier as private.

Creation of Objects:
Once the class is created, one or more objects can be created from the class as objects are instance of the class.
Juts as we declare a variable of data type int as:
int x;
Objects are also declared as:
class name followed by object name;
exforsys e1;
This declares e1 to be an object of class exforsys.
For example a complete class and object declaration is given below:

```
class exforsys
{
private:
int x,y;
public:
void sum()
{
……
……
}
};
main()
{
exforsys e1;
……
……
}
```

The object can also be declared immediately after the class definition. In other words the object name can also be placed immediately before the closing flower brace symbol } of the class declaration.

For example

```
class exforsys
{
private:
int x,y;
public:
void sum()
{
……
……
}
}e1 ;
```

The above code also declares an object **e1** of class exforsys.
It is important to understand that in object-oriented programming language, when a class is created no memory is allocated. It is only when an object is created is memory then allocated.

Function Overloading

A function is overloaded when same name is given to different function. However, the two

functions with the same name will differ at least in one of the following.

a) The number of parameters
b) The data type of parameters
c) The order of appearance

These three together are referred to as the **function signature**.

For example if we have two functions :

void foo(int i,char a);
void boo(int j,char b);

Their signature is the same **(int ,char)** but a function

void moo(int i,int j) ; has a signature **(int, int)** which is different.
While **overloading a function**, the return type of the functions needs to be the same.

In general functions are overloaded when :
1. Functions differ in function signature.
2. Return type of the functions is the same.

Here s a basic example of **function overloading**

```
#include <iostream>
using namespace std;
class arith {
public:
void calc(int num1)
{
cout<<lSquare of a given number: — <<num1*num1 <<endl;
}
void calc(int num1, int num2 )
{
cout<<lProduct of two whole numbers: — <<num1*num2 <<endl;
}
};
int main() //begin of main function
{
arith a;
a.calc(5);
a.calc(6,7);
}
```

Let us see what we did in the **function overloading example**.

First the overloaded function in this example is **calc**. If you have noticed we have in our **arith** class two functions with the name **calc**. The fist one takes one integer number as a parameter and prints the square of the number. The second calc function takes two integer numbers as parameters, multiplies the numbers and prints the product. This is all we need for making a successful overloading of a function.

a) we have two functions with the same name : calc
b) we have different signatures : (int) , (int, int)
c) return type is the same : void

The result of the execution looks like this

Square of a given number: 25
Product of two whole numbers: 42
The result demonstrates the overloading concept. Based on the arguments we use when we call the **calc** function in our code :

```
a.calc(5);
a.calc(6,7);
```

The compiler decides witch function to use at the moment we call the function.

C++ Friend Functions

Need for Friend Function

As discussed in the earlier sections on access specifiers, when a data is declared as private inside a class, then it is not accessible from outside the class. A function that is not a member or an external class will not be able to access the private data. A programmer may have a situation where he or she would need to access private data from non-member functions and external classes. For handling such cases, the concept of Friend functions is a useful tool.

What is a Friend Function?

A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class.

How to define and use Friend Function in C++?

The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. The friend function must have the class to which it is declared as friend passed to it in argument.

Some important points to note while using friend functions in C++:

- 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, seen in example below.

Example to understand the friend function:

```
#include <iostream.h>
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+e2.b)-5;
}

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

The output of the above program is
The result is:295

The function compute() is a non-member function of the class exforsys. In order to make this function have access to the private data a and b of class exforsys , it is created as afriend function for the class exforsys. As a first step, the function compute() is declared as friend in the class exforsys as:

friend int compute (exforsys e1)

The keyword friend is placed before the function. The function definition is written as a normal function and thus, the function has access to the private data a and b of the class exforsys. It is declared as friend inside the class, the private data values a and b are added, 5 is subtracted from the result, giving 295 as the result. This is returned by the function and thus the output is displayed as shown above.

Constant and volatile member functions

A member function declared with the **const** qualifier can be called for constant and nonconstant objects. A nonconstant member function can only be called for a nonconstant object. Similarly, a member function declared with the **volatile** qualifier can be called for volatile and nonvolatile objects. A nonvolatile member function can only be

called for a nonvolatile object.

static members

Class members can be declared using the storage class specifier static in the class member list. Only one copy of the static member is shared by all objects of a class in a program. When you declare an object of a class having a static member, the static member is not part of the class object.

A typical use of static members is for recording data common to all objects of a class. For example, you can use a static data member as a counter to store the number of objects of a particular class type that are created. Each time a new object is created, this static data member can be incremented to keep track of the total number of objects.

You access a static member by qualifying the class name using the :: (scope resolution) operator. In the following example, you can refer to the static member f() of class type X as X::f() even if no object of type X is ever declared:

```
class X
{
static int f();
};

int main() {
X::f();
}
```

Pointers to classes

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example:

```
CRectangle * prect;
```

is a pointer to an object of class CRectangle.

As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (->) of indirection. Here is an example with some possible combinations:

```
// pointer to classes example
#include <iostream>
using namespace std;

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

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

```
int main () {
CRectangle a, *b, *c;
CRectangle * d = new CRectangle[2];
b= new CRectangle;
c= &a;
a.set_values (1,2);
b->set_values (3,4);
d->set_values (5,6);
d[1].set_values (7,8);
cout << —a area: — << a.area() << endl;
cout << —*b area: — << b->area() << endl;
cout << —*c area: — << c->area() << endl;
cout << —d[0] area: — << d[0].area() << endl;
cout << —d[1] area: — << d[1].area() << endl;
delete[] d;
delete b;
return 0;
}
```

Output:
a area: 2
*b area: 12
*c area: 2
d[0] area: 30
d[1] area: 56

Next you have a summary on how can you read some pointer and class operators (*, &, ., ->, [ ]) that appear in the previous example:

Expression Can be read as

*x	pointed by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed by x
(*x).y	member y of object pointed by x (equivalent to the previous one)
X[0]	first object pointed by x
X[1]	second object pointed by x
X[n]	(n+1)th object pointed by x

Difference between const variables and const object
Constant variables are the variables whose value cannot be changed throughout the program but if any object is constant, value of any of the data members (const or non const) of that object cannot be changed throughout the program. Constant object can invoke only constant function.

Nested classes
A nested class is declared within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables.

Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class. The following example demonstrates this:

```
class A {
int x;
class B { };
class C {
// The compiler cannot allow the following
// declaration because A::B is private:
// B b;

int y;
void f(A* p, int i) {

// The compiler cannot allow the following
// statement because A::x is private:
// p->x = i;

}
};

void g(C* p) {
// The compiler cannot allow the following
// statement because C::y is private:
// int z = p->y;
}
};

int main() { }
```

The compiler would not allow the declaration of object b because class A::B is private. The compiler would not allow the statement p->x = i because A::x is private. The compiler would not allow the statement int z = p->y because C::y is private.

Local classes

A local class is declared within a function definition. Declarations in a local class can only use type names, enumerations, static variables from the enclosing scope, as well as external variables and functions.

For example:

```
int x;
void f()
{
static int y;
                        // global variable
                        // function definition
                        // static variable y can be used by
// local class
int x;      // auto variable x cannot be used by
// local class
extern int g();   // extern function g can be used by
// local class
class local
```

```
    {
// local class
int g() { return x; }
    // error, local variable x

    // cannot be used by g
    int h() { return y; }   // valid,static variable y
    int k() { return ::x; } // valid, global x
    int l() { return g(); } // valid, extern function g
    };
    }

    int main()
    {
    local* z;
    // error: the class local is not visible
    // ...}
```

Member functions of a local class have to be defined within their class definition, if they are defined at all. As a result, member functions of a local class are inline functions. Like all member functions, those defined within the scope of a local class do not need the keyword inline.

A local class cannot have static data members. In the following example, an attempt to define a static member of a local class causes an error:

```
void f()
{
class local
{
int f();
// error, local class has noninline
// member function
int g() {return 0;} // valid, inline member function
static int a;    // error, static is not allowed for
// local class
int b;       // valid, nonstatic variable
};
}
//  . . .
```

An enclosing function has no special access to members of the local class.

UNIT 2

Constructors and Destructors In C++

Constructors:

What is the use of Constructor

The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.

General Syntax of Constructor

Constructor is a special member function that takes the same name as the class name. The syntax generally is as given below:

<class name> { arguments};

The default constructor for a class X has the form

X::X()

In the above example the arguments is optional.

The constructor is automatically invoked when an object is created.

The various types of constructors are

- Default constructors
- Parameterized constructors
- Copy constructors

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(); //default
Constructor
...
};

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

Parameterized Constructor:

A parameterized constructor is just one that has parameters specified in it.

Example:

```
class Exforsys
{
private:
int a,b;
public:
Exforsys(int,int);// Parameterized constructor
...
```

```
};

Exforsys :: Exforsys(int x, int y)
{
a=x;
b=y;
}
```

Copy constructor;

One of the more important forms of an overloaded constructor is the copy constructor.
The purpose of the copy constructor is to initialize a new object with data copied from another object of the same class.

For example to invoke a copy constructor the programmer writes:

```
Exforsys e3(e2);
```
or
```
Exforsys e3=e2;
```

Both the above formats can be used to invoke a copy constructor.

For Example:

```
#include <iostream.h>
class Exforsys()
{
private:
int a;
public:
Exforsys()
{}
Exforsys(int w)
{
a=w;
}
Exforsys(Exforsys& e)
{
a=e.a;
cout<<l Example of Copy
Constructorl;
}
void result()
{
cout<< a;
}
};

void main()
{
Exforsys e1(50);
Exforsys e3(e1);
```

```
cout<< —\ne3=l;e3.result();
}
```

In the above the copy constructor takes one argument an object of type Exforsys which is passed by reference. The output of the above program is

```
Example of Copy Constructor
e3=50
```

Some important points about constructors:

- 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, volatile, or const volatile.
- No return type is specified for a constructor.
- The constructor must be defined in the public. The constructor must be a public member.
- Overloading of constructors is possible.

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 called 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.

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.

For example:

```
class Exforsys
{
private:
……
```

```
public:
Exforsys()
{}
~Exforsys()
{}
}
```

Operator Overloading

Operator overloading is a very important feature of Object Oriented Programming. It is because by using this facility programmer would be able to create new definitions to existing operators. In other words a single operator can perform several functions as desired by programmers.

Operators can be broadly classified into:

Unary Operators

Binary Operators

Unary Operators:

As the name implies takes operate on only one operand. Some unary operators are namely

++ Increment operator

-- Decrement operator

- Unary minus

Operators:

The arithmetic operators, comparison operators, and arithmetic assignment operators come under this category.

Both the above classification of operators can be overloaded. So let us see in detail each of this.

Operator Overloading – Unary operators

As said before operator overloading helps the programmer to define a new functionality for the existing operator. This is done by using the keyword **operator**.

The general syntax for defining an operator overloading is as follows:


```
return_type classname :: operator operator symbol(argument)
{
……..
statements;
}
```

Thus the above clearly specifies that operator overloading is defined as a member function by making use of the keyword operator.

In the above:

return_type – is the data type returned by the function

class name - is the name of the class
operator – is the keyword
operator symbol – is the symbol of the operator which is being overloaded or defined for new functionality
:: - is the scope resolution operator which is used to use the function definition outside the class.

For example

Suppose we have a class say Exforsys and if the programmer wants to define a operator overloading for unary operator say ++, the function is defined as

Inside the class Exforsys the data type that is returned by the overloaded operator is defined as

```
class Exforsys
{
private:
.....
public:
void operator ++( );
.....
};
```

The important steps involved in defining an operator overloading in case of unary operators are namely
27
Inside the class the operator overloaded member function is defined with the return data type as member function or a friend function.
 If the function is a member function then the number of arguments taken by the operator member function is none.
 If the function defined for the operator overloading is a friend function then it takes one argument.

Now let us see how to use this overloaded operator member function in the program

```
#include <iostream.h>
class Exforsys
{
private:
int x;
public:
Exforsys( ) { x=0; }
void display();
void Exforsys ++( );
//Constructor
//overload unary ++
};
void Exforsys :: display()
{
```

```
cout<<l\nValue of x is: — << x;
}

void Exforsys :: operator ++( ) //Operator Overloading for operator ++ defined
{
++x;
}
void main( )
{
Exforsys e1,e2;
//Object e1 and e2 created
cout<<lBefore Incrementl
cout <<l\nObject e1: l<<e1.display();
cout <<l\nObject e2: l<<e2.display();
++e1; //Operator overloading applied
++e2;
cout<<l\n After Incrementl
cout <<l\nObject e1: l<<e1.display();
cout <<l\nObject e2: l<<e2.display();
}
```

The output of the above program is:

```
Before Increment
Object e1:
Value of x is: 0
Object e1:
Value of x is: 0
Before Increment
Object e1:
Value of x is: 1
Object e1:
Value of x is: 1
```

In the above example we have created 2 objects e1 and e2 f class Exforsys. The operator ++ is overloaded and the function is defined outside the class Exforsys.

When the program starts the constructor Exforsys of the class Exforsys initialize the values as zero and so when the values are displayed for the objects e1 and e2 it is displayed as zero. When the object ++e1 and ++e2 is called the operator overloading function gets applied and thus value of x gets incremented for each object separately. So now when the values are displayed for objects e1 and e2 it is incremented once each and gets printed as one for each object e1 and e2.

Operator Overloading – Binary Operators

Binary operators, when overloaded, are given new functionality. The function defined for binary operator overloading, as with unary operator overloading, can be member function or friend function.

The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one (see below example). When the function defined for the binary operator overloading is a friend function, then it uses two arguments.

Binary operator overloading, as in unary operator overloading, is performed using a keyword operator.

Binary operator overloading example:

```
#include <iostream.h>
class Exforsys
{
private:
int x;
int y;

public:
Exforsys()
{ x=0; y=0; }

void getvalue( )
//Constructor
//Member Function for Inputting Values
{
cout << —\n Enter value for x: —;
cin >> x;
cout << —\n Enter value for y: —;
cin>> y;
}

void displayvalue( )

//Member Function for Outputting Values
{
cout <<lvalue of x is: — << x <<l; value of y is: —<<y
}

Exforsys operator +(Exforsys);
};

Exforsys Exforsys :: operator + (Exforsys e2)
//Binary operator overloading for + operator defined
{
int x1 = x+ e2.x;
int y1 = y+ e2.y;
return Exforsys(x1,y1);
}

void main( )
{
Exforsys e1,e2,e3;
//Objects e1, e2, e3 created
```