

Constructors

A constructor is a 'special' member function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
// class with a constructor

class integer
{
    int m, n;
    public:
        integer(void);           // constructor declared
        .....
        .....
};
integer :: integer(void)        // constructor defined
{
    m = 0; n = 0;
}
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```
integer int1;           // object int1 created
```

not only creates the object **int1** of type **integer** but also initializes its data members **m** and **n** to zero. There is no need to write any statement to invoke the constructor function (as we do with the normal member functions). If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately. This would be very inconvenient, if there are a large number of objects.

A constructor that accepts no parameters is called the *default constructor*. The default constructor for **class A** is **A::A()**. If no such constructor is defined, then the compiler supplies a default constructor. Therefore a statement such as

```
A a;
```

invokes the default constructor of the compiler to create the object **a**.

Constructor in C++



Default Constructors

Default constructor is the constructor which doesn't take any argument. It has no parameters.

```
// Cpp program to illustrate the
// concept of Constructors
#include <iostream>
using namespace std;

class construct {
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }
};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl
         << "b: " << c.b;
}
```

Parametrized Constructors

The constructor `integer()`, defined above, initializes the data members of all the objects to zero. However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called *parameterized constructors*.

The constructor **integer()** may be modified to take arguments as shown below:

```
class integer
{
    int m, n;
public:
    integer(int x, int y);  // parameterized constructor
    .....
    .....
};
integer :: integer(int x, int y)
{
    m = x; n = y;
}
```


When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer(0,100); // explicit call
```

This statement creates an integer object `int1` and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);           // implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Remember, when the constructor is parameterized, we must provide appropriate arguments for the constructor.

```
// CPP program to illustrate
// parameterized constructors
#include <iostream>
using namespace std;

class Point {
private:
    int x, y;

public:
    // Parameterized Constructor
    Point(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    int getX()
    {
        return x;
    }
    int getY()
    {
        return y;
    }
};

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}
```

The constructor functions have some special characteristics. These are :

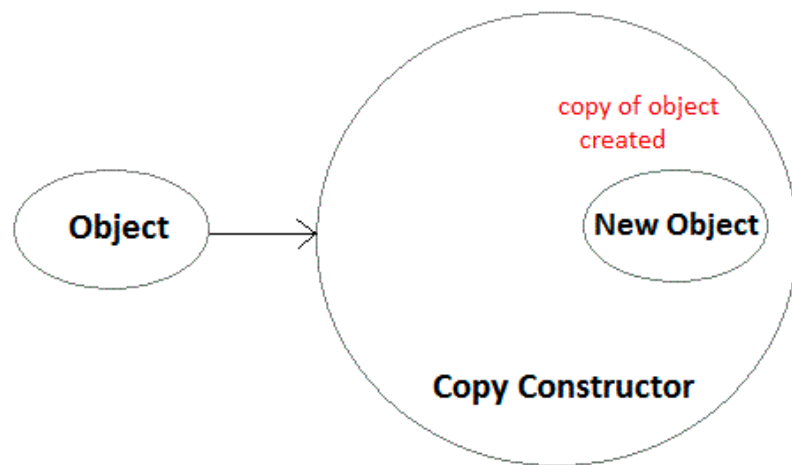
- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**. (Meaning of virtual will be discussed later in Chapter 9.)
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the operators **new** and **delete** when memory allocation is required.

Remember, when a constructor is declared for a class, initialization of the class objects becomes mandatory.

Copy Constructor

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```



The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to –

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.

```
#include<iostream>
using namespace std;
class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1) { x = x1; y = y1; }

    // Copy constructor
    Point(const Point &p1) {x = p1.x; y = p1.y; }

    int getX()          { return x; }
    int getY()          { return y; }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
    Point p2 = p1; // Copy constructor is called here

    // Let us access values assigned by constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

    return 0;
}
```


Constructor Overloading

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to **function overloading**.

- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

Constructor with Default Arguments

It is possible to define constructors with default arguments. For example, the constructor `complex()` can be declared as follows:

```
complex(float real, float imag=0);
```

The default value of the argument **imag** is zero. Then, the statement

```
complex C(5.0);
```

assigns the value 5.0 to the **real** variable and 0.0 to **imag** (by default). However, the statement

```
complex C(2.0,3.0);
```

assigns 2.0 to **real** and 3.0 to **imag**. The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor **A::A()** and the default argument constructor **A::A(int = 0)**. The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

```
A a;
```

The ambiguity is whether to 'call' **A::A()** or **A::A(int = 0)**.

Destructors

A *destructor*, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor for the class `integer` can be defined as shown below:

```
~integer(){ }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever **new** is used to allocate memory in the constructors, we should use **delete** to free that memory. For example, the destructor for the **matrix** class discussed above may be defined as follows:

```
matrix :: ~matrix()
{
    for(int i=0; i<d1; i++)
        delete p[i];
    delete p;
}
```

```
#include <iostream>
using namespace std;
class HelloWorld{
public:
    //Constructor
    HelloWorld(){
        cout<<"Constructor is called"<<endl;
    }
    //Destructor
    ~HelloWorld(){
        cout<<"Destructor is called"<<endl;
    }
    //Member function
    void display(){
        cout<<"Hello World!"<<endl;
    }
};
int main(){
    //Object created
    HelloWorld obj;
    //Member function called
    obj.display();
    return 0;
}
```


Dynamic Constructors

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator.


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

class show
{
    const char* p;

public:
    // default constructor
    show()
    {
        // allocating memory at run time
        p = new char[6];
        p = "show";
    }

    void display()
    {
        cout << p << endl;
    }
};

int main()
{
    show obj = show();
    obj.display();
}
```

Operator Overloading

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

- The meaning of an operator is always same for variable of basic types like: int, float, double etc. For example: To add two integers, + operator is used.
- However, for user-defined types (like: objects), you can redefine the way operator works. For example:

If there are two objects of a class that contains string as its data members. You can redefine the meaning of + operator and use it to concatenate those strings.

This feature in C++ programming that allows programmer to redefine the meaning of an operator (when they operate on class objects) is known as operator overloading.

How to Overload an operator??

To overload an operator, a special operator function is defined inside the class as:

```
class className
{
    ... ..
    public
        returnType operator symbol (arguments)
        {
            ... ..
        }
    ... ..
};
```

- Here, return type is the return type of the function. The return type of the function is followed by operator keyword.
- Symbol is the operator symbol you want to overload. Like: +, <, -, ++
- You can pass arguments to the operator function in similar way as functions.

Example-Unary Operator Overloading

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

class Test
{
    int count;

public:
    Test()
    {
        count=5;
    }

    void operator ++()
    {
        count = count+1;
    }
    void Display() { cout<<"Count: "<<count; }
};

int main()
{
    Test t;
    // this calls "function void operator ++()" function
    ++t;
    t.Display();
    return 0;
}
```

Example-Binary Operator Overloading

```
#include<iostream>
using namespace std;
class test
{
    int a,b;
public:
    test()
    {
        a=0;
        b=0;
    }
    test(int x, int y)
    {
        a=x;
        b=y;
    }
    test operator+(test t);
    void disp();
};

test test::operator+(test t)
{
    test temp;
    temp.a=a+t.a;
    temp.b=b+t.b;
    return temp;
}
```

```
void test::disp()  
[ {  
    cout<<"value of a"<<a<<endl;  
    cout<<"value of b"<<b<<endl;  
- }  
int main()  
[ {  
    test t1,t2,t3;  
    t1=test(10,20);  
    t2=test(30,40);  
    t3=t1+t2;  
    t1.disp();  
    t2.disp();  
    t3.disp();  
    return 0;  
- }
```


Things to Remember

- Operator overloading allows you to redefine the way operator works for user-defined types only (objects, structures). It cannot be used for built-in types (int, float, char etc.).
- Two operators = and & are already overloaded by default in C++. For example: To copy objects of same class, you can directly use = operator. You do not need to create an operator function.
- Operator overloading cannot change the precedence and associativity of operators. However, if you want to change the order of evaluation, parenthesis should be used.
- There are 4 operators that cannot be overloaded in C++. They are :: (scope resolution), . (member selection), .* (member selection through pointer to function) and ?: (ternary operator).

Operator Overloading using Friend Functions

C++ has tried to bridge-in the gap between the user-defined data-type variables and built-in data-type variables, by allowing us to work on both type of variables in a same way by the concept of **operator overloading**.

- Operator overloading can be achieved in two ways -By an **operator overloading by member function**, or
- By an operator overloading *non-member* **friend** function

Overloading binary minus operator - using friend function

As we know that the minus operator - when applied to any built-in type variable such as int, float, double, long will change its value from positive to negative. We can even change the sign of values of an object by using the unary - operator using **member function**. But let's see how to achieve the same using a *non-member* **friend function**.

```

#include<iostream>

using namespace std;

class A
{
int a;

public:
void set_a();
void get_a();
friend A operator -(A); // Friend function which takes an object of A and return an object of A type.
};
//Definition of set_a() function
void A :: set_a()
{
a = 10;
}
//Definition of get_a() function
void A :: get_a()
{
cout<< a <<"\n";
}
//Definition of overloaded unary minus operator - friend function
A operator -(A ob)
{
ob.a = -(ob.a);
return ob;
}

int main()
{
A ob;
ob.set_a();
cout<<"The value of a is : ";
ob.get_a();
//Calling operator overloaded function - to negate the value
ob = -ob; //ob object is passed as an argument to the friend function and its negative version is returned.
cout<<"The value of a after calling operator overloading friend function - is : ";
ob.get_a();
}

```

Why Friend Functions?

friend Function in C++

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the **function**.

The compiler knows a given function is a friend function by the use of the keyword **friend**.

For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword `friend`.

Declaration of friend function in C++

```
class class_name
{
    ... ..
    friend return_type function_name(argument/s);
    ... ..
}
```

Now, you can define the friend function as a normal function to access the data of the class. No `friend` keyword is used in the definition.

- Friend function using operator overloading offers better flexibility to the class.
- These functions are not a members of the class.
- When you overload a unary operator you have to pass one argument.
- When you overload a binary operator you have to pass two arguments.
- Friend function can access private members of a class directly.

Pointers to Objects

Just like other pointers, the object pointers are declared by placing in front of a object pointer's name. It takes the following general form :

```
class-name * object-pointer ;
```

where class-name is the name of an already defined class and object-pointer is the pointer to an object of this class type. For example, to declare optr as an object pointer of Sample class type, we shall write

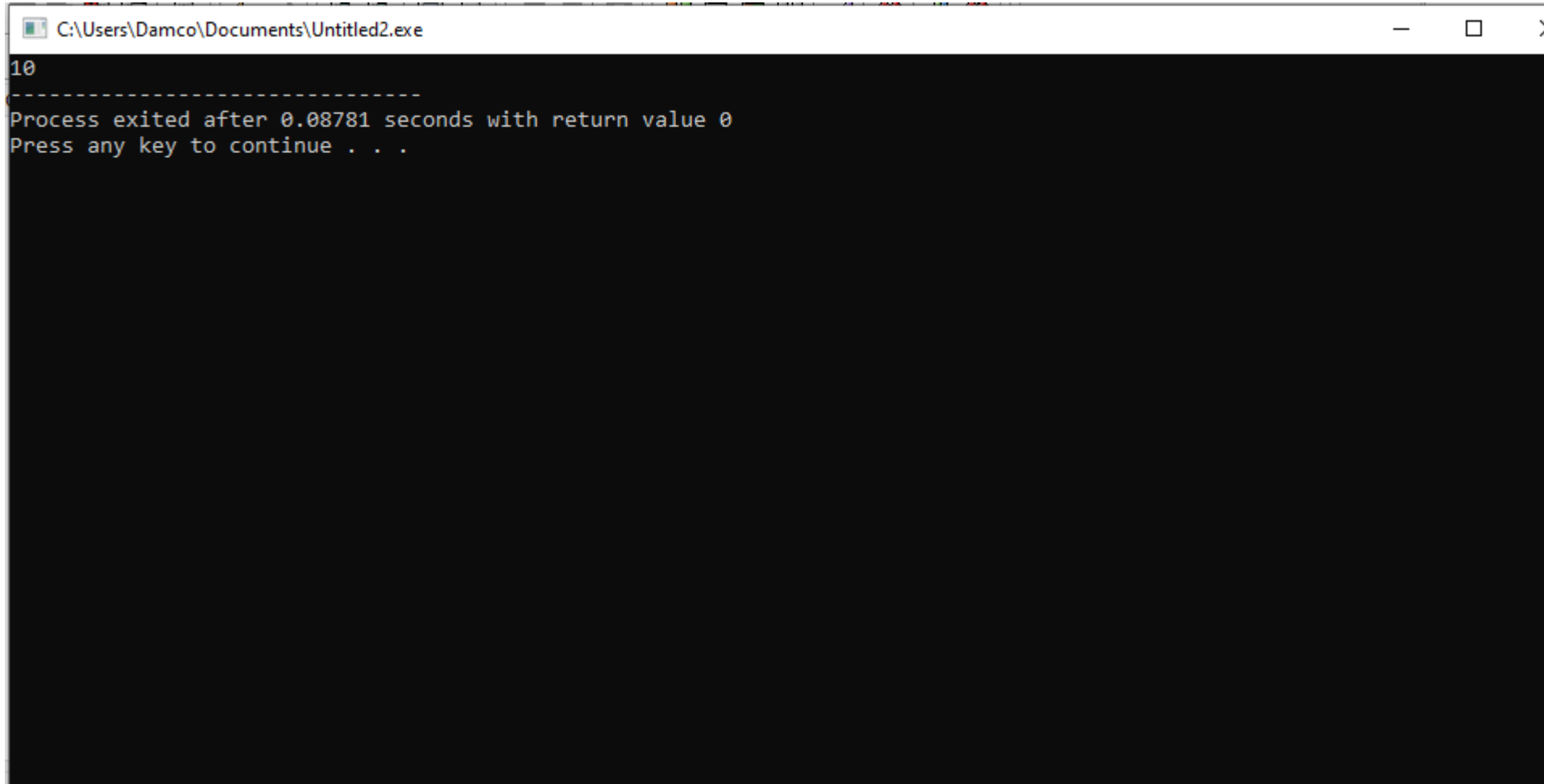
```
Sample *optr ;
```

where Sample is already defined class. When accessing members of a class using an object pointer, the arrow

```
#include<iostream>
using namespace std;
class myclass
{
    int i;
public:
    void read(int j)
    {
        i = j;
    }
    int getint()
    {
        return i;
    }
};

int main()
{
    myclass ob, *objectPointer;
    objectPointer = &ob; // get address of ob
    objectPointer->read(10);
    cout<<objectPointer->getint(); // use -> to call getint()
    return 0;
}
```

Output:



A screenshot of a Windows command prompt window. The title bar at the top shows the file path "C:\Users\Damco\Documents\Untitled2.exe" and standard window controls (minimize, maximize, close). The command prompt area has a black background with white text. The output displayed is: "10", followed by a dashed line separator, then "Process exited after 0.08781 seconds with return value 0", and finally "Press any key to continue . . .".

```
C:\Users\Damco\Documents\Untitled2.exe
10
-----
Process exited after 0.08781 seconds with return value 0
Press any key to continue . . .
```

This pointer

A Variable That Holds an Address value is called a Pointer variable or simply pointer. We already discussed about pointer that's point to Simple data types likes int, char, float etc. So similar to these type of data type, Objects can also have an address, so there is also a pointer that can point to the address of an Object, This Pointer is Known as **This Pointer**.

THIS POINTER:

Every Object in C++ has access to its own address through an important pointer called **This** Pointer. The **This** Pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object. Whenever a member function is called, it is automatically passed an implicit arguments that is This pointer to the invoking object (*i.e. The object*

The `This` pointer is passed as a hidden argument to all Nonstatic member function calls and is available as a local variable within the body of all Nonstatic functions. `This` Pointer is a constant pointer that holds the memory address of the current object. `This` pointer is not available in static member functions as static member functions can be called without any object (*with class name*).

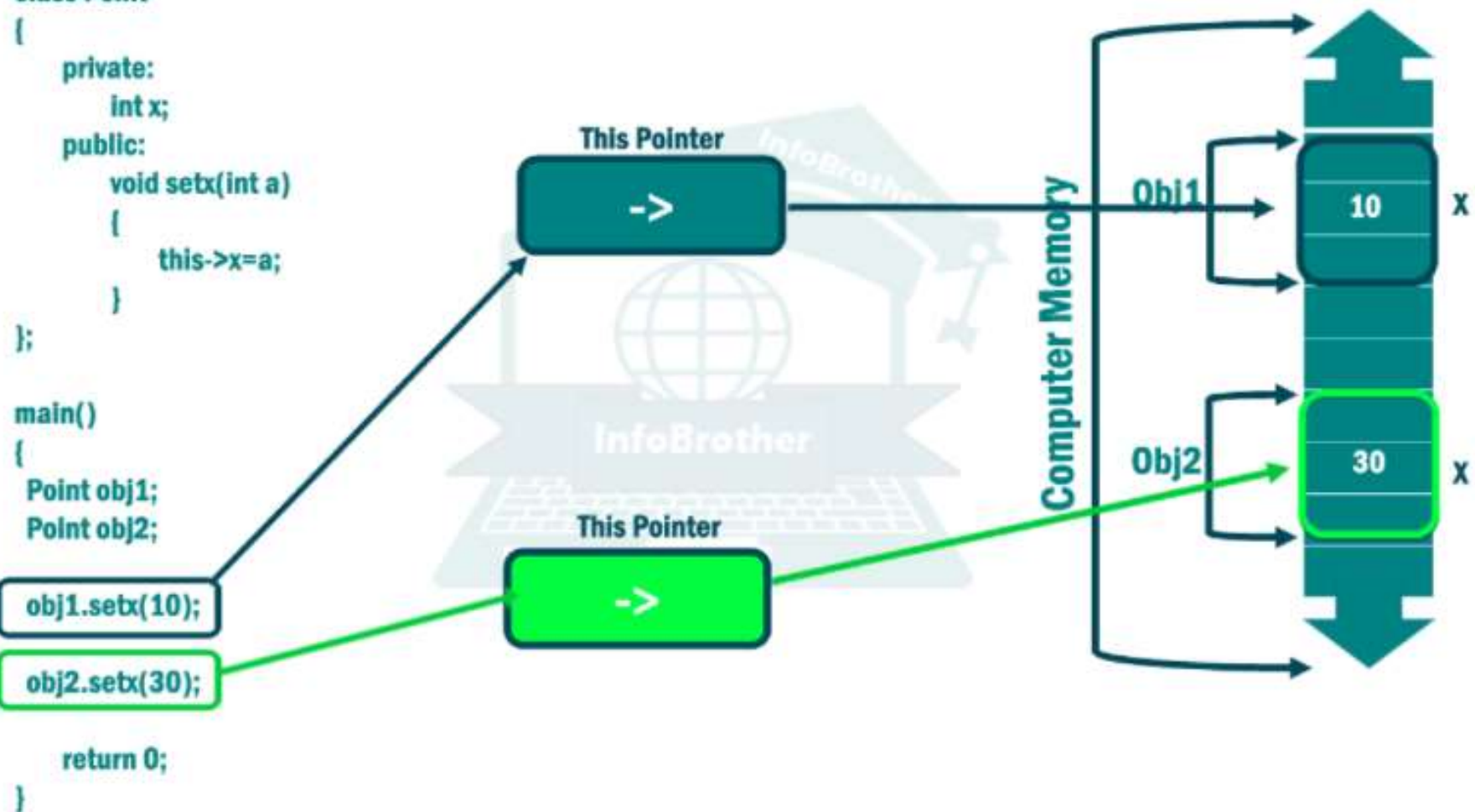
```
class Point
{
    private:
        int x;
    public:
        void setx(int a)
        {
            this->x=a;
        }
};
```

```
main()
{
    Point obj1;
    Point obj2;
```

```
    obj1.setx(10);
```

```
    obj2.setx(30);
```

```
    return 0;
```



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

/* Local variable is same as a member's name */
class Test
{
private:
int x;
public:
void setX (int x)
{
    // The 'this' pointer is used to retrieve the object's x
    // hidden by the local variable 'x'
    this->x = x;
}
void print()
{
cout << "x = " << x << endl;
}
};

int main()
{
Test obj;
int x = 20;
obj.setX(x);
obj.print();
return 0;
}
```