

# 'this' pointer in C++

To understand 'this' pointer, it is important to know that how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share single copy of member functions.

Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?

Compiler supplies an implicit pointer along with the functions names as 'this'.

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).

Following are the situations where 'this' pointer is used:

## 1) When local variable's name is same as member's name

```
#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;
}
```

Output:

```
x = 20
```

## 2) To return reference to the calling object

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
    // Some processing
    return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

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

class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test &setX(int a) { x = a; return *this; }
    Test &setY(int b) { y = b; return *this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1(5, 5);


    // Chained function calls. All calls modify the same object
    // as the same object is returned by reference
    obj1.setX(10).setY(20);

    obj1.print();
    return 0;
}
```

Output:

```
x = 10 y = 20
```



## Question 1



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


class Test
{
private:
    int x;
public:
    Test(int x = 0) { this->x = x; }
    void change(Test *t) { this = t; }
    void print() { cout << "x = " << x << endl; }
};

int main()
{
    Test obj(5);
    Test *ptr = new Test (10);
    obj.change(ptr);
    obj.print();
    return 0;
}
```



```
prog.cpp:10:29: error: lvalue required as left operand
void change(Test *t) { this = t; }
                        ^
```



## Question 2



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





class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    static void fun1() { cout << "Inside fun1()"; }
    static void fun2() { cout << "Inside fun2()"; this->fun1(); }
};

int main()
{
    Test obj;
    obj.fun2();
    return 0;
}
```



```
cpp:12:47: error: 'this' is unavailable for static member
```

### Question 3







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

class Test
{
private:
    int x;
    int y;
public:
    Test (int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test setX(int a) { x = a; return *this; }
    Test setY(int b) { y = b; return *this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1;
    obj1.setX(10).setY(20);
    obj1.print();
    return 0;
}
```

> x = 10 y = 0

### Question 4



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

class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    void setX(int a) { x = a; }
    void setY(int b) { y = b; }
    void destroy() { delete this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj;
    obj.destroy();
    obj.print();
    return 0;
}
```



## Abort signal from abort(3) (SIGABRT)

SIGABRT is commonly used by libc and other libraries to abort the program in case of critical errors. For example, glibc sends an SIGABRT in case of a detected double-free or other heap corruptions.

Also, most "assert" implementations make use of SIGABRT in case of a failed assert.

Furthermore, SIGABRT can be sent from any other process like any other signal. Of course, the sending process needs to run as same user or root.

It usually happens when there is a problem with memory allocation.

It happened to me when my program was trying to allocate an array with negative size.

abort() sends the calling process the SIGABRT signal, this is how abort() basically works.

abort() is usually called by library functions which detect an internal error or some seriously broken constraint. For example malloc() will call abort() if its internal structures are damaged by a heap overflow.

---

In C++, scope resolution operator is ::. It is used for following purposes.

**1) To access a global variable when there is a local variable with same name:**



```
// C++ program to show that we can access a global variable
// using scope resolution operator :: when there is a local
// variable with same name
#include<iostream>
using namespace std;

int x; // Global x

int main()
{
    int x = 10; // Local x
    cout << "Value of global x is " << ::x;
    cout << "\nValue of local x is " << x;
    return 0;
}
```

Value of global x is 0

Value of local x is 10

## 2) To define a function outside a class.

```
// C++ program to show that scope resolution operator :: is used
// to define a function outside a class
#include<iostream>
using namespace std;

class A
{
public:
    // Only declaration
    void fun();
};

// Definition outside class using ::
void A::fun()
{
    cout << "fun() called";
}

int main()
{
    A a;
    a.fun();
    return 0;
}
```

Output:

```
fun() called
```

## 3) To access a class's static variables.

```
// C++ program to show that :: can be used to access static
// members when there is a local variable with same name
#include<iostream>
using namespace std;

class Test
{
    static int x;
public:
    static int y;

    // Local parameter 'a' hides class member
    // 'a', but we can access it using ::
    void func(int x)
    {
        // We can access class's static variable
        // even if there is a local variable
        cout << "Value of static x is " << Test::x;

        cout << "\nValue of local x is " << x;
    }
};
```

```

int Test::x = 1;
int Test::y = 2;

int main()
{
    Test obj;
    int x = 3 ;
    obj.func(x);

    cout << "\nTest::y = " << Test::y;

    return 0;
}

```

Output:

```

Value of static x is 1
Value of local x is 3
Test::y = 2;

```

#### 4) In case of multiple Inheritance:

If same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

```

// Use of scope resolution operator in multiple inheritance.
#include<iostream>
using namespace std;

class A
{
protected:
    int x;
public:
    A() { x = 10; }
};

class B
{
protected:
    int x;
public:
    B() { x = 20; }
};

class C: public A, public B
{
public:
    void fun()
    {
        cout << "A's x is " << A::x;
        cout << "\nB's x is " << B::x;
    }
};

```

```
int main()
{
    C c;
    c.fun();
    return 0;
}
```

Output:

```
A's x is 10
B's x is 20
```

---

Consider below C++ program:

```
// C++ program to show that local parameters hide
// class members
#include<iostream>
using namespace std;
class Test
{
    int a;
public:
    Test() { a = 1; }

    // Local parameter 'a' hides class member 'a'
    void func(int a) { cout << a; }
};

int main()
{
    Test obj;
    int k = 3 ;
    obj.func(k);
    return 0;
}
```


Output:

```
3
```

The output for the above program is 3 since the “a” passed as argument to the “func” shadows the “a” of the class .i.e 1

Then how to output the class's ‘a’. This is where **this pointer** comes in handy. A statement like “cout <a” instead of “cout << a” can simply output the value 1 as this pointer points to the object from whom func is called.





```
// C++ program to show use of this to access member when
// there is a local variable with same name.
#include<iostream>
using namespace std;
class Test
{
    int a;
public:
    Test() { a = 1; }


    // Local parameter 'a' hides object's member
    // 'a', but we can access it using this.
    void func(int a) { cout << this->a; }
};

int main()
{
    Test obj;
    int k = 3 ;
    obj.func(k);
    return 0;
}
```

Output:

1

**How about Scope Resolution Operator?** We cannot use Scope resolution operator in above example to print object's member 'a' because scope resolution operator can only be used for a static data member (or class members). If we use scope resolution operator in above program we get compiler error and if we use this pointer in below program, then also we get compiler error.



```
// C++ program to show that :: can be used to access static
// members when there is a local variable with same name
#include<iostream>
using namespace std;
class Test
{
    static int a; // a IS STATIC NOW
public:

    // Local parameter 'a' hides class member
    // 'a', but we can access it using ::
    void func(int a) { cout << Test::a; }
};
```

```
// In C++, static members must be explicitly defined
// like this
int Test::a = 1;

int main()
{
    Test obj;
    int k = 3 ;
    obj.func(k);
    return 0;
}
```

Output:

1

*Conclusion is scope resolution operator is for accessing static or class members and this pointer is for accessing object members when there is a local variable with same name.*