This pointer

The `this` pointer in C++ is a special pointer that all non-static member functions have. It points to the object for which the member function was called. This is useful in several scenarios:

1. **To access the object's members**: When a member function is called, `this` is automatically set to the address of the object that the function was called on. This allows the function to access the object's members. This is especially useful when a function's parameters have the same names as the object's members, as in provided code:

```
Person(std::string name, std::string ID) {
    this->name = name;
    this->ID = ID;
}
```

Here, `this->name` refers to the `name` member of the object, and `name` refers to the function's parameter. Without `this`, there would be a naming conflict.
2. **To return the object itself from a member function**: Sometimes, you might want a member function to return the object it was called on. You can do this by returning `*this`.
3. **To check if two objects are the same**: If you have a member function that takes another object of the same type as a parameter, you can use `this` to check if the parameter refers to the same object as the one the function was called on.

Remember, `this` is a pointer, so you need to use `->` to access members, and `*this` to refer to the object itself.

Keyword Const

You can not assign inside a const function

```
    void display_person_info(void) const{
        std::cout << "Person name is " << name << std::endl;
```

```
        std::cout << "Person ID is " << ID << std::endl;
        /* @attention: Error, assignation in const function */
        //name = "Safia";
    }
```

Also you can not call non constant function for constant object

```
        ..
        ..
    void display_person_info(void) {
        std::cout << "Person name is " << name << std::endl;
        std::cout << "Person ID is " << ID << std::endl;
        /* @attention: Error, assignation in const function */
        //name = "Safia";
    }
};
int main(void)
{
    const Person p1{"Safia","123"};
    p1.display_person_info();
    /*Error Assignment(Compiler does not know if you will change member inside)
     of read-only member 'Person::name'*/
    return 0;
}
```

Keyword Static

```
void func_test()
{
    static int test=0;
    test++;
    std::cout << "Test is " << test << std::endl;
```

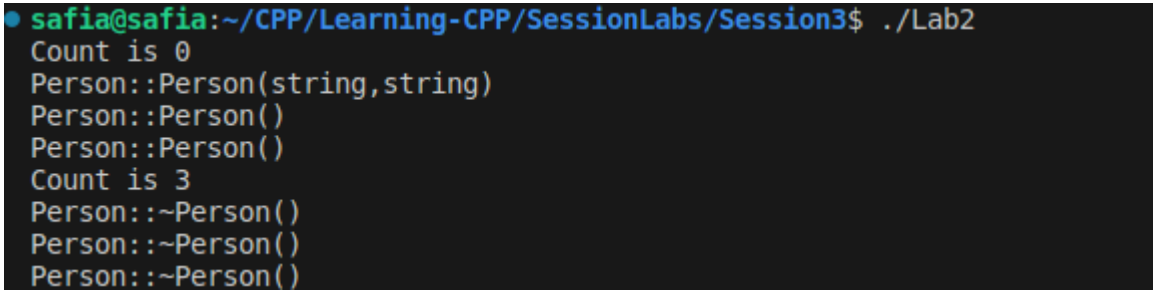
```
}
int main(void)
{
    func_test();
    func_test();
    func_test();
    return 0;
}
```

Static member inside class (Shared Resource)

```
class Person
{
    std::string name;
    std::string ID;
    public:
    //static member variable shared by all objects of the class not specific to any object
    //Can't initialize in the class
     static int count;
     ..
     ..
     ..
};
//static member variable initialization
int Person::count = 0;
int main(void)
{
    std::cout << "Count is " << Person::count << std::endl;
    Person p1{"Safia","123"};
    Person p2;
    Person p3;
```

```
    std::cout << "Count is " << Person::count << std::endl;
}
```

Output



Static Function

Can be called without creating any object of the class

```
        ..
        ..
    //static member function
    //Static function can only access static members
    static void display_count(void){
        std::cout << "Count is " << count << std::endl;
    }
};
//static member variable initialization
int Person::count = 0;
int main(void)
{
        //Can be called without creating any object of the class
    Person::display_count();
}
```

Friend

The friend keyword in C++ is used to grant a function or another class access to private and protected members of a class.

Normally, these members are not accessible from outside the class, but sometimes you might have a function or a class that needs to access them. This is where friend comes in.
In provided code, func_test is declared as a friend function of the Person class:
This means that func_test can access the private members name and ID of Person objects. Without the friend keyword, func_test would not be able to do this.

Remember, the friend keyword breaks the encapsulation principle of object-oriented programming, so it should be used sparingly and only when necessary. It's often better to provide public methods that control access to private members in a controlled way.

```
class Person
{
    std::string name;
    std::string ID;
    //friend function can access private members of the class attributes and methods
    friend void func_test();
        ..
        ..
}
```

Inheritance

Polymorphism

Operator Overloading

Using the traditional operator to work with user defined data types as the built in data types
Traditional

```
class Number
{
    int x;
    public:
        void Multiply(int mul)
        {
            x *= mul;
        }
        void divide (int div)
        {
            x /= div;
        }
};

int main()
{
    Number n1;
    n1.Multiply(10);
    n1.divide(5);
    return 0;
}
```

User defined data types
But i need to do it more easier like `n1+10` or `n1/5`

First: Assignment

`Person2 = Person1;` :All member values in person1 assigned to person2

```
        ..
        ..
        ..
```

```
    //Assignment operator overloading
    //To implement deep copy of the object, the return type should be reference to the object
    //chained equality like p1=p2=p3 can not be done if the return type is void because the return type of the
first assignment will be void

    Person &operator=(const Person &source){
        std::cout << "Person::operator=(const Person &)" << std::endl;
        if(this == &source){
            return *this;
        }
        name = source.name;
        ID = source.ID;
        return *this;
    }
};
int main(void)
{
    Person p1{"Safia","1234"},p2;
    p2 = p1 ;
    p2.display_person_info();
    return 0;
}
```

Examble 2

Assignment (copy) operator and move operator

```
class myString
{
    char *str;
    public:
    //default constructor
    myString():str{nullptr}{
        str = new char('\0');
```

```
    }
    //parameterized constructor
    myString(char *src)
    {
        str = new char[strlen(src)+1];
        strcpy(str,src);
    }
    //Copy constructor
    myString(const myString &source){
        str = new char[strlen(source.str)+1];
        strcpy(str,source.str);
    }
    //Assignment operator overloading
    myString &operator=(const myString &source){
        //check for self assignment
        if(this == &source){
            return *this;
        }
        //delete the memory allocated for the str
        delete [] str;
        //allocate memory for the new str
        str = new char[strlen(source.str)+1];
        //copy the source string to the str
        strcpy(str,source.str);
        //return the reference to the object
        return *this;
    }
    ~myString(){
        delete [] str;
    }

};
int main(void)
{
```

```
    Person p1{"Safia","1234"},p2;
    p2 = p1 ;
    p2.display_person_info();

    myString obj{"Hello"};
    myString str{obj}; //Copy constructor
    myString str3;
    str3 = str; //Assignment operator overloading
    return 0;
}
int main(void)
{
    myString obj{"Hello"};
    myString str{obj}; //Copy constructor
    myString str3;
    str3 = str; //Assignment operator overloading
    //move constructor
    myString str4{std::move(str)};
    //move assignment operator overloading
    myString str5;
    str5 = std::move(str3);

    return 0;
}
```