

# - Examples

In this lesson, we'll learn about the examples of class templates.

## WE'LL COVER THE FOLLOWING ^

- Example 1: Templates in Class
  - Explanation
- Example 2: Inheritance in Class Templates
  - Explanation
- Example 3: Methods in Class Templates
  - Explanation

## Example 1: Templates in Class #

```
// templateClassTemplate.cpp

#include <iostream>

class Account{
public:
    explicit Account(double amount=0.0): balance(amount){}

    void deposit(double amount){
        balance+= amount;
    }

    void withdraw(double amount){
        balance-= amount;
    }

    double getBalance() const{
        return balance;
    }

private:
    double balance;
};

template <typename T, int N>
class Array{
```

```

public:
    Array()= default;
    int getSize() const;

private:
    T elem[N];
};

template <typename T, int N>
int Array<T,N>::getSize() const {
    return N;
}

int main(){

    std::cout << std::endl;

    Array<double,10> doubleArray;
    std::cout << "doubleArray.getSize(): " << doubleArray.getSize() << std::endl;

    Array<Account,1000> accountArray;
    std::cout << "accountArray.getSize(): " << accountArray.getSize() << std::endl;

    std::cout << std::endl;
}

```



## Explanation #

We have created two `Array` class objects, i.e., `doubleArray` and `accountArray` in lines 45 and 48. By calling generic function `getSize()` in line 37, we can access the size of different objects.

## Example 2: Inheritance in Class Templates #

```

// templateClassTemplateInheritance.cpp

#include <iostream>

template <typename T>
class Base{
public:
    void func1() const {
        std::cout << "func1()" << std::endl;
    }
    void func2() const {
        std::cout << "func2()" << std::endl;
    }
    void func3() const {
        std::cout << "func3()" << std::endl;
    }
};

```



```

template <typename T>
class Derived: public Base<T>{
public:

    using Base<T>::func2;

    void callAllBaseFunctions(){
        this->func1();
        func2();
        Base<T>::func3();
    }
};

int main(){
    std::cout << std::endl;

    Derived<int> derived;
    derived.callAllBaseFunctions();

    std::cout << std::endl;
}

```



## Explanation #

We have implemented both a `Base` and a `Derived` class. `Derived` is publicly inherited from `Base` and may, therefore, use in its method `callAllBaseFunctions` in line 24, the methods `func1`, `func2`, and `func3` from the `Base` class.

- **Make the name dependent:** The call `this->func1` in line 25 is dependent. The name lookup will consider in this case all base classes.
- **Introduce the name into the current scope:** The expression `using Base<T>::func2` (line 22) introduces `func2` into the current scope.
- **Call the name fully qualified:** Calling `func3` fully qualified (line 27) will break a virtual dispatch and may cause new surprises.

We have created a `Derived` class object named `derived`. By using this object, we can access the base class functions by calling the method `callAllBaseFunctions`.

## Example 3: Methods in Class Templates #

```
// templateClassTemplateMethods.cpp
```



```

#include <algorithm>
#include <iostream>
#include <vector>

template <typename T, int N>
class Array{

public:
    Array()= default;

    template <typename T2>
    Array<T, N>& operator=(const Array<T2, N>& arr){
        elem.clear();
        elem.insert(elem.begin(), arr.elem.begin(), arr.elem.end());
        return *this;
    }

    int getSize() const;

    std::vector<T> elem;
};

template <typename T, int N>
int Array<T, N>::getSize() const {
    return N;
}

int main(){

    Array<double, 10> doubleArray{};
    Array<int, 10> intArray{};

    doubleArray= intArray;

    Array<std::string, 10> strArray{};
    Array<int, 100> bigIntArray{};

    // doubleArray= strArray;           // ERROR: cannot convert 'const std::basic_string<char>' to 'const Array<double, 10>'
    // doubleArray= bigIntArray;        // ERROR: no match for 'operator=' in 'doubleArray = bigIntArray'
}

```



## Explanation #

In the example above, we have initialized two instances of `Array` class namely `doubleArray` and `intArray` in lines 32 and 33. We're using the generic `=` operator to copy the `intArray` elements to `doubleArray` in line 35. When you look carefully, you see that the generic `=` is only applicable, when both arrays have the same length.

In the next lesson, we have a small challenge for you to test your knowledge of class templates

class templates.