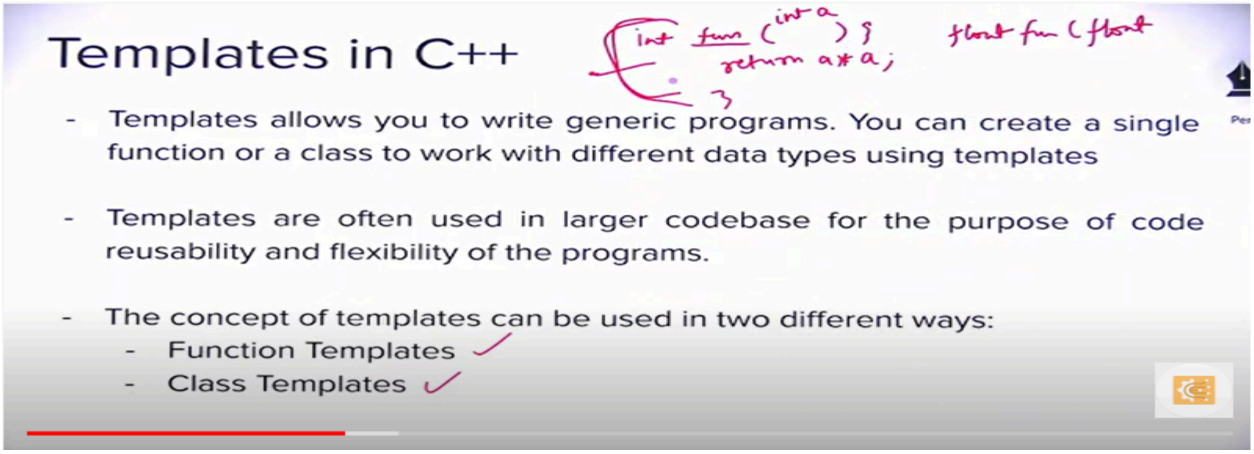
Templates in C++ (lec-34)

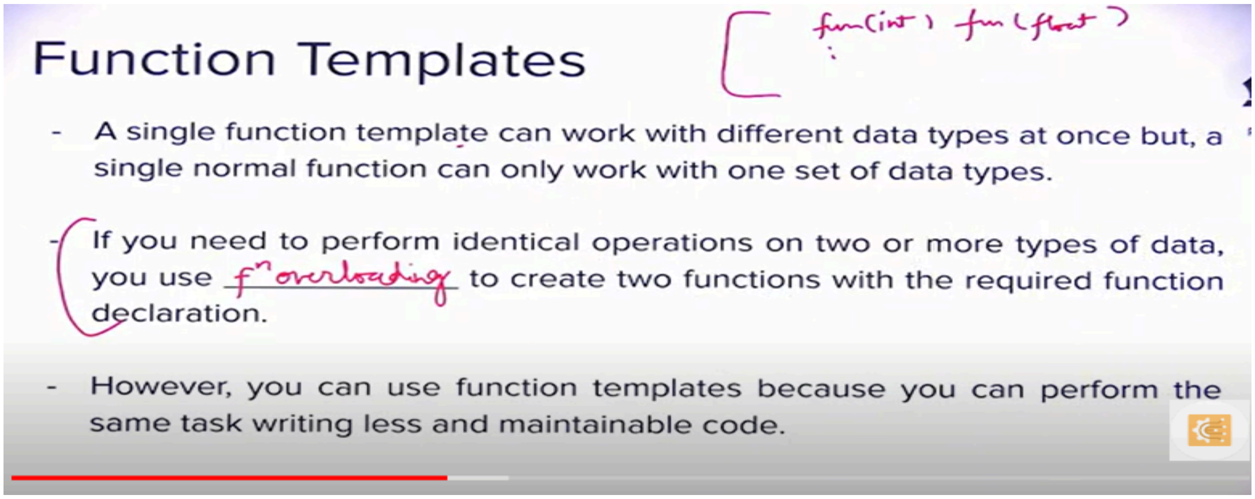
—-----------------------



Templates in C++

- Templates allows you to write generic programs. You can create a single function or a class to work with different data types using templates
- Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.
- The concept of templates can be used in two different ways:
  - Function Templates ✓
  - Class Templates ✓

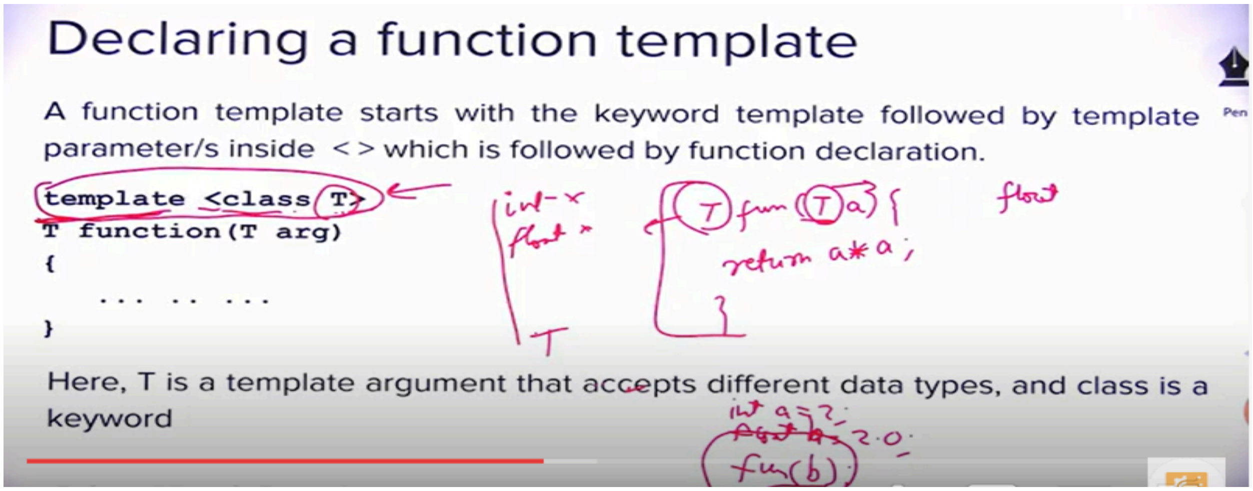
....



Function Templates

- A single function template can work with different data types at once but, a single normal function can only work with one set of data types.
- If you need to perform identical operations on two or more types of data, you use fⁿ overloading to create two functions with the required function declaration.
- However, you can use function templates because you can perform the same task writing less and maintainable code.

....



Declaring a function template

A function template starts with the keyword template followed by template parameter/s inside < > which is followed by function declaration.

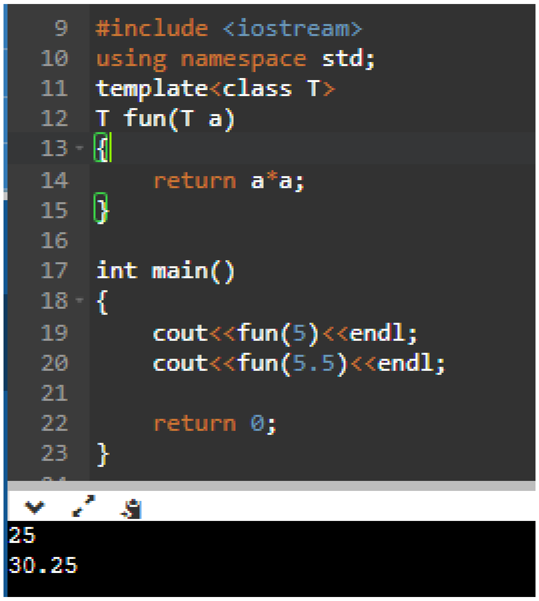
```
template <class T>
T function(T arg)
{
    ... .. ...
}
```

Here, T is a template argument that accepts different data types, and class is a keyword

Function Template

—----------------------------------

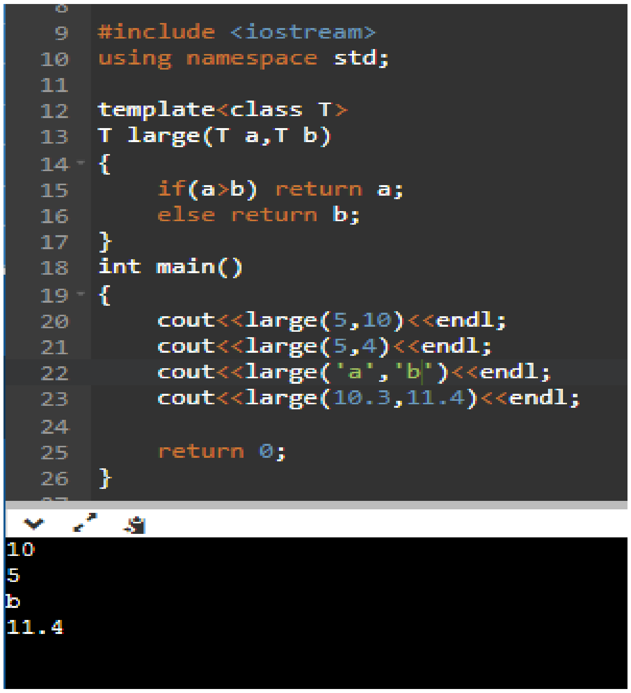
Example 1:



```cpp
#include <iostream>
using namespace std;
template<class T>
T fun(T a)
{
    return a*a;
}

int main()
{
    cout<<fun(5)<<endl;
    cout<<fun(5.5)<<endl;

    return 0;
}
```

```
25
30.25
```

Example 2:



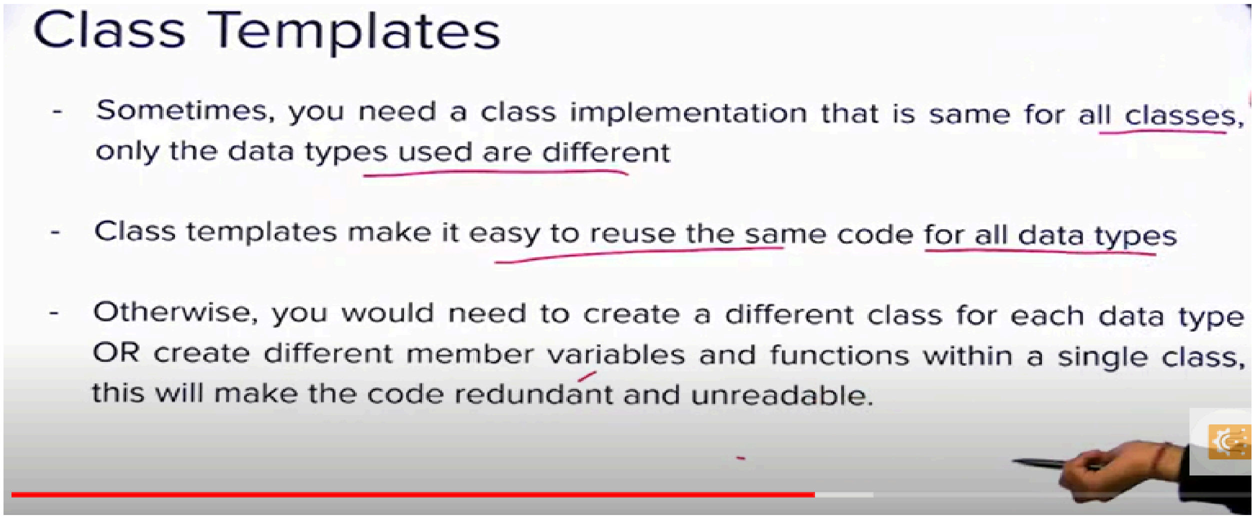
```cpp
#include <iostream>
using namespace std;

template<class T>
T large(T a,T b)
{
    if(a>b) return a;
    else return b;
}
int main()
{
    cout<<large(5,10)<<endl;
    cout<<large(5,4)<<endl;
    cout<<large('a','b')<<endl;
    cout<<large(10.3,11.4)<<endl;

    return 0;
}
```

```
10
5
b
11.4
```

Class Template:

—-----------------------

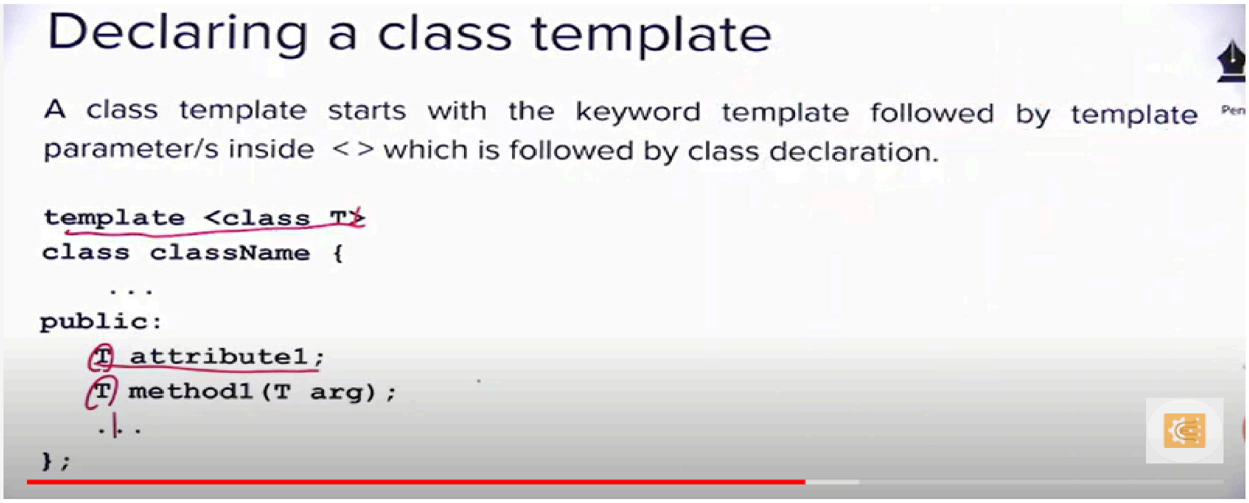
Note: constructor can be overloaded.



Class Templates

- Sometimes, you need a class implementation that is same for all classes, only the data types used are different
- Class templates make it easy to reuse the same code for all data types
- Otherwise, you would need to create a different class for each data type OR create different member variables and functions within a single class, this will make the code redundant and unreadable.

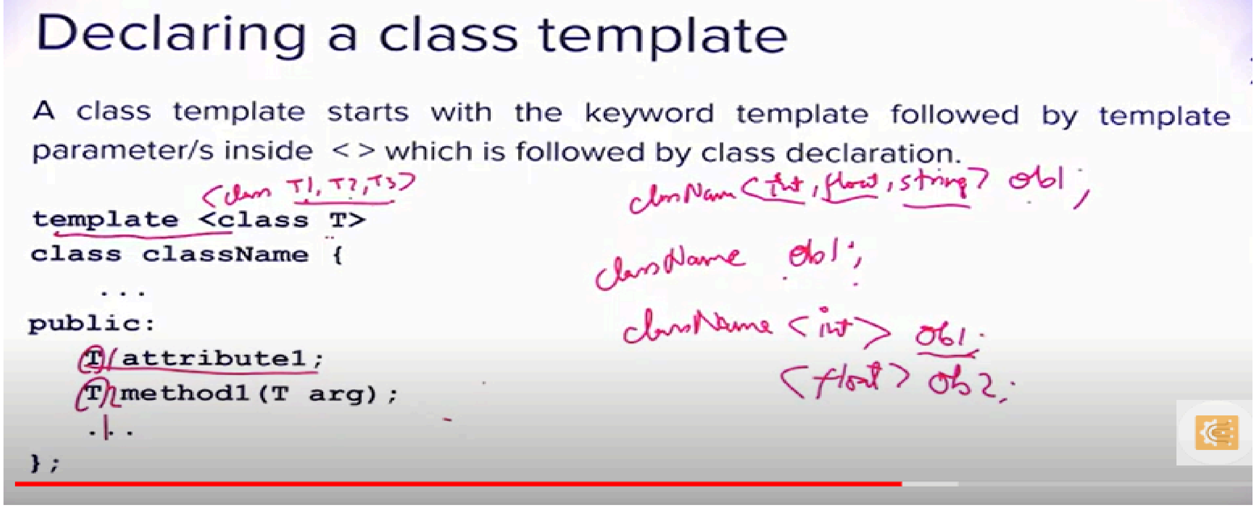
………….



Declaring a class template

A class template starts with the keyword template followed by template parameter/s inside < > which is followed by class declaration.

```
template <class T>
class className {
    ...
public:
    T attribute1;
    T method1(T arg);
    ...
};
```



Declaring a class template

A class template starts with the keyword template followed by template parameter/s inside < > which is followed by class declaration.

```
template <class T>
class className {
    ...
public:
    T attribute1;
    T method1(T arg);
    ...
};
```

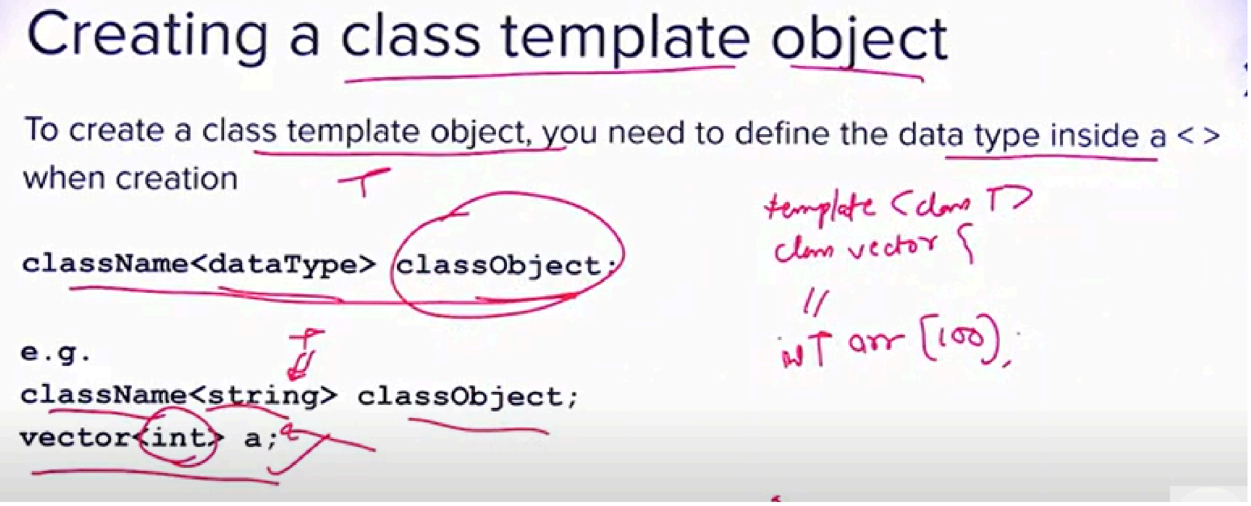
<class T1, T2, T3>

className<int, float, string> ob1;

className ob1;

className <int> ob1.

<float> ob2;



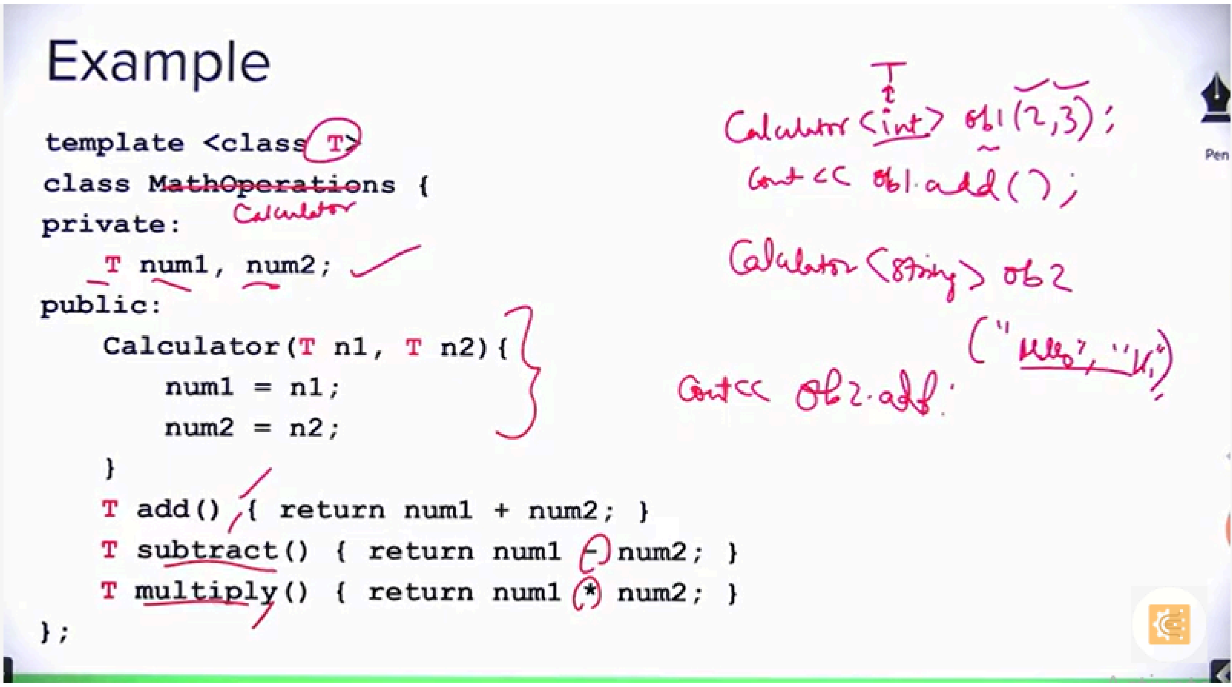
Creating a class template object

To create a class template object, you need to define the data type inside a < > when creation

```
className<dataType> classObject;

e.g.
className<string> classObject;
vector<int> a;
```

```
template <class T>
class vector {
  //
  int T arr [100];
```

Example:



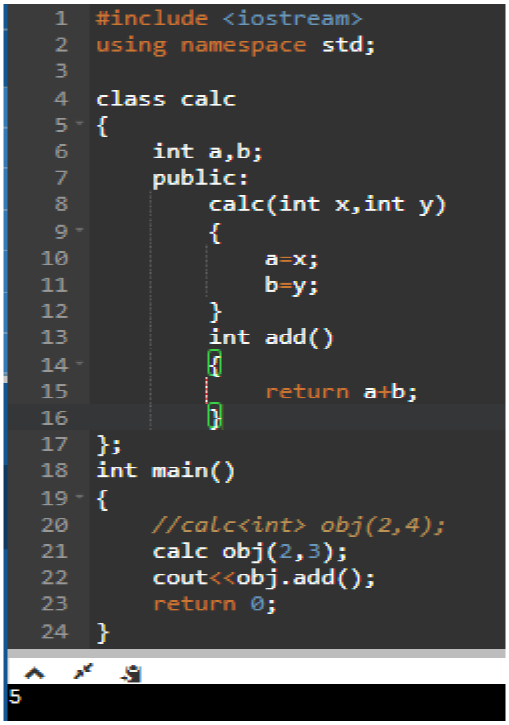
Example

```
template <class T>
class ~~MathOperations~~ {
      Calculator
private:
    T num1, num2;
public:
    Calculator(T n1, T n2){
        num1 = n1;
        num2 = n2;
    }
    T add() { return num1 + num2; }
    T subtract() { return num1 - num2; }
    T multiply() { return num1 * num2; }
};
```

```
Calculator<int> ob1(2,3);
Cout << ob1.add();

Calculator<string> ob2 ("Hello", "Hi");
Cout<< ob2.add.
```

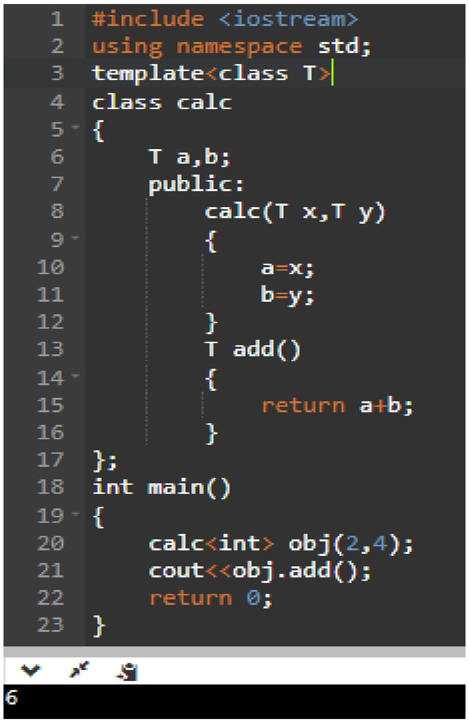
Example in template:

—---------------------------



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

class calc
{
    int a,b;
    public:
        calc(int x,int y)
        {
            a=x;
            b=y;
        }
        int add()
        {
            return a+b;
        }
};
int main()
{
    //calc<int> obj(2,4);
    calc obj(2,3);
    cout<<obj.add();
    return 0;
}
```

```
5
```



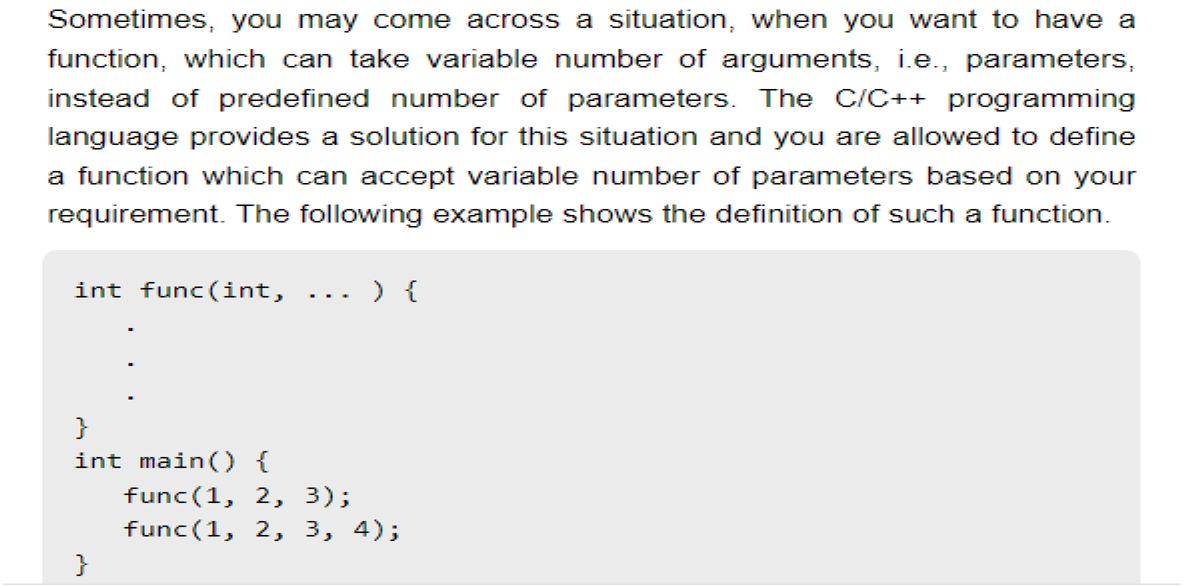
```cpp
#include <iostream>
using namespace std;
template<class T>
class calc
{
    T a,b;
    public:
        calc(T x,T y)
        {
            a=x;
            b=y;
        }
        T add()
        {
            return a+b;
        }
};
int main()
{
    calc<int> obj(2,4);
    cout<<obj.add();
    return 0;
}
```

```
6
```

Variable number of arguments in C++

—---

https://www.tutorialspoint.com/variable-number-of-arguments-in-cplusplus



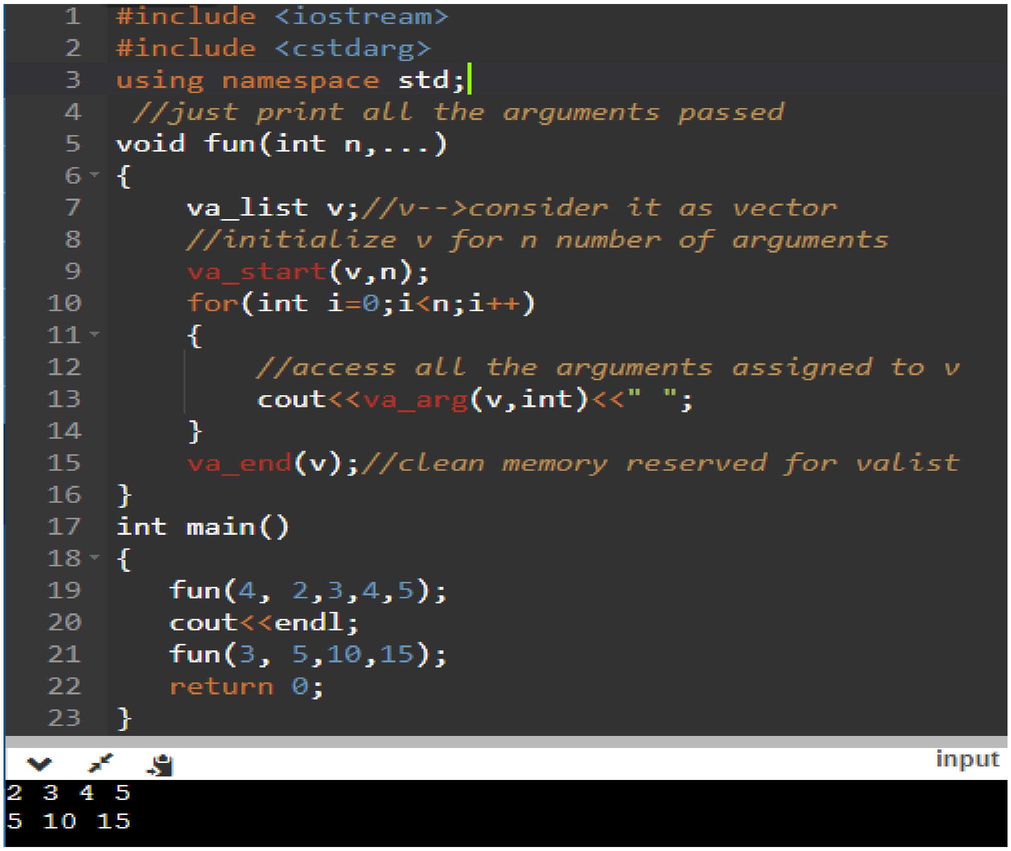
Sometimes, you may come across a situation, when you want to have a function, which can take variable number of arguments, i.e., parameters, instead of predefined number of parameters. The C/C++ programming language provides a solution for this situation and you are allowed to define a function which can accept variable number of parameters based on your requirement. The following example shows the definition of such a function.

```
int func(int, ... ) {
   .
   .
   .
}
int main() {
   func(1, 2, 3);
   func(1, 2, 3, 4);
}
```

It should be noted that the function func() has its last argument as **ellipses,** i.e. three dotes (...) and the one just before the ellipses is always an int which will represent the total number variable arguments passed. To use such functionality, you need to make use of stdarg.h header file which provides the functions and macros to implement the functionality of variable arguments and follow the given steps –

- Define a function with its last parameter as ellipses and the one just before the ellipses is always an int which will represent the number of arguments.
- Create a **va_list** type variable in the function definition. This type is defined in stdarg.h header file.
- Use int parameter and **va_start** macro to initialize the **va_list** variable to an argument list. The macro **va_start** is defined in stdarg.h header file.
- Use **va_arg** macro and **va_list** variable to access each item in argument list.
- Use a macro **va_end** to clean up the memory assigned to **va_list** variable.

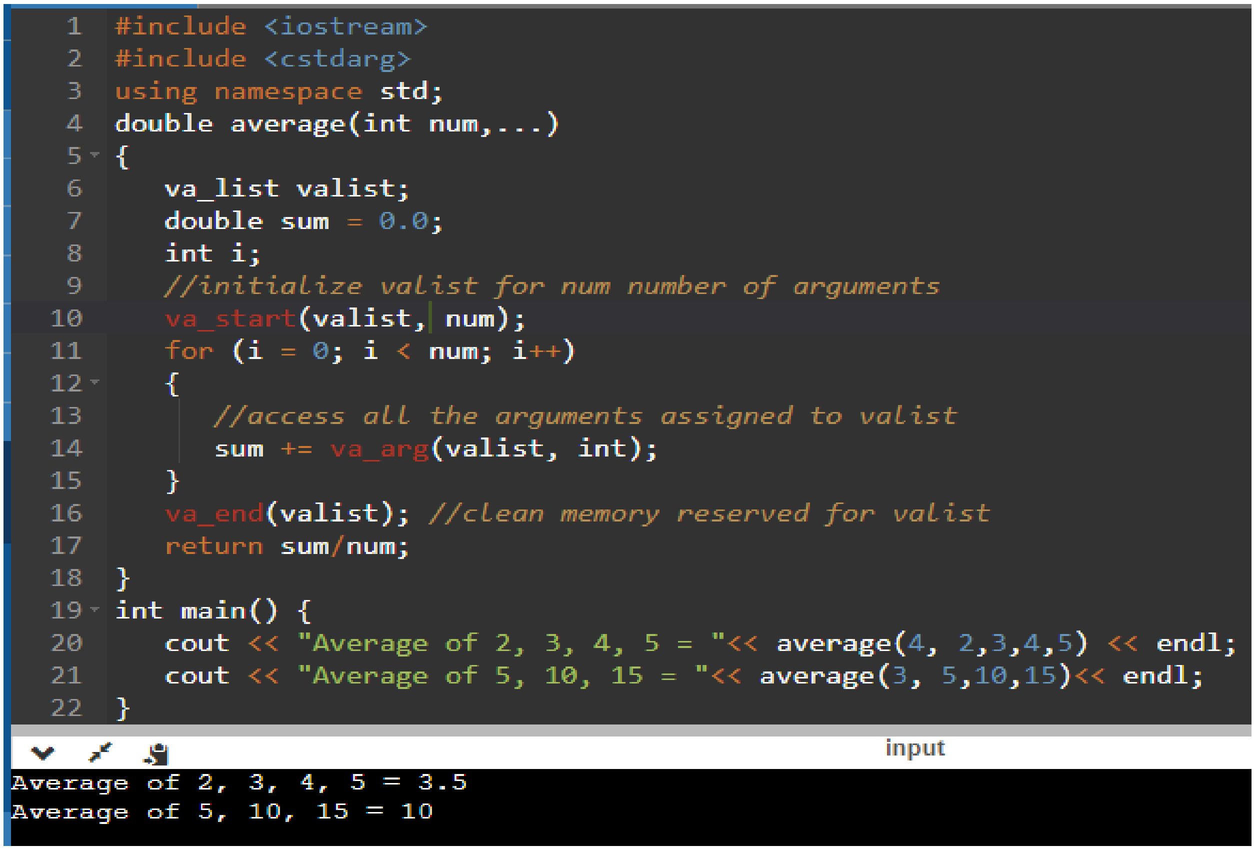
Q:take the multiple number of arguments and print.



```cpp
#include <iostream>
#include <cstdarg>
using namespace std;
 //just print all the arguments passed
void fun(int n,...)
{
    va_list v;//v-->consider it as vector
    //initialize v for n number of arguments
    va_start(v,n);
    for(int i=0;i<n;i++)
    {
        //access all the arguments assigned to v
        cout<<va_arg(v,int)<<" ";
    }
    va_end(v);//clean memory reserved for valist
}
int main()
{
   fun(4, 2,3,4,5);
   cout<<endl;
   fun(3, 5,10,15);
   return 0;
}
```

```
2 3 4 5
5 10 15
```

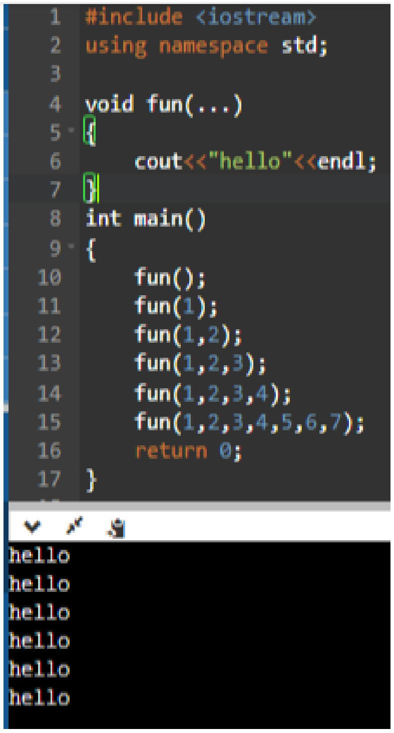
Q:Now let us follow the above steps and write down a simple function which can take the variable number of parameters and return their average –



```cpp
#include <iostream>
#include <cstdarg>
using namespace std;
double average(int num,...)
{
    va_list valist;
    double sum = 0.0;
    int i;
    //initialize valist for num number of arguments
    va_start(valist, num);
    for (i = 0; i < num; i++)
    {
        //access all the arguments assigned to valist
        sum += va_arg(valist, int);
    }
    va_end(valist); //clean memory reserved for valist
    return sum/num;
}
int main() {
    cout << "Average of 2, 3, 4, 5 = "<< average(4, 2,3,4,5) << endl;
    cout << "Average of 5, 10, 15 = "<< average(3, 5,10,15)<< endl;
}
```

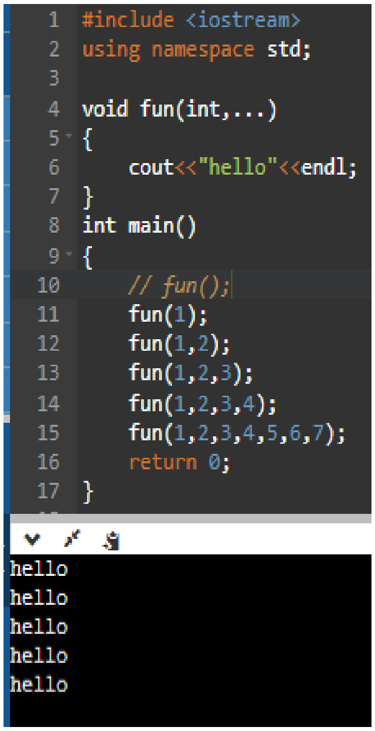
```
Average of 2, 3, 4, 5 = 3.5
Average of 5, 10, 15 = 10
```

Example:



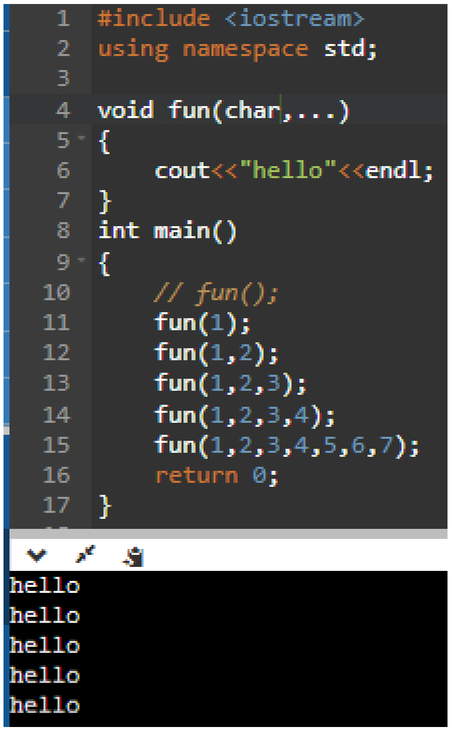
```cpp
#include <iostream>
using namespace std;

void fun(...)
{
      cout<<"hello"<<endl;
}
int main()
{
      fun();
      fun(1);
      fun(1,2);
      fun(1,2,3);
      fun(1,2,3,4);
      fun(1,2,3,4,5,6,7);
      return 0;
}
```

```
hello
hello
hello
hello
hello
hello
```



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

void fun(int,...)
{
      cout<<"hello"<<endl;
}
int main()
{
      // fun();
      fun(1);
      fun(1,2);
      fun(1,2,3);
      fun(1,2,3,4);
      fun(1,2,3,4,5,6,7);
      return 0;
}
```

```
hello
hello
hello
hello
hello
```



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

void fun(char,...)
{
      cout<<"hello"<<endl;
}
int main()
{
      // fun();
      fun(1);
      fun(1,2);
      fun(1,2,3);
      fun(1,2,3,4);
      fun(1,2,3,4,5,6,7);
      return 0;
}
```

```
hello
hello
hello
hello
hello
```